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, który reprezentowałem w rozgrywkach Młodej Ligi, przeznaczonych dla zawodników poniżej 23 roku życia, oraz w rozgrywkach akademickich. 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 .....

Streszczenie

Praca prezentuje projekt biblioteki zawierającej uniwersalny silnik służący do tworzenia biurowych wykresów w Qt oraz Qt Quick. Wykresy te 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. These plots 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. 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. Qt posiada już kilka bibliotek służących tworzeniu wykresów, jednak nadal istnieje zapotrzebowanie na wartościowe rozwiązanie open-source, co wykażę w rozdziale *Przegląd dziedziny*.

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.

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.

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*. 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.

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

— *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* [2]. 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,
— JavaScript – imperatywny język,
— Środowisko uruchomieniowe zintegrowane z Qt,
— Designer,
— API C++ umożliwiające integrację z aplikacjami Qt.

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* [3].

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 [4] 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.

[2] Framework Graphics View `http://qt-project.org/doc/qt-5.0/qtwidgets/graphicsview.html`
[3] Scene Graph `http://en.wikipedia.org/wiki/Scene_graph`
[4] Qt Build Suit `http://qt-project.org/wiki/qbs`

Ciekawą cechą QML jest system wiązania wartości ze zmienną, który umożliwia uzależnienie zmiennej A od zmiennej B. Zmiana wartości zmiennej B w czasie wykonywania programu spowoduje automatyczne przeliczenie wartości zmiennej 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

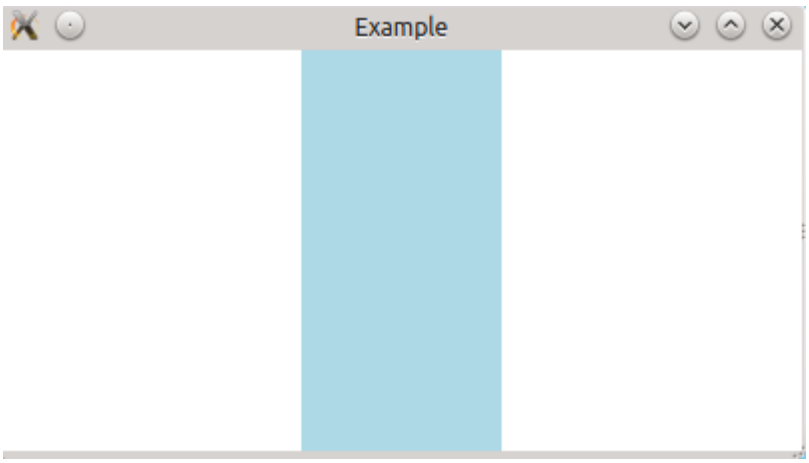
Wydruk 1.1. Przykład QML

```
Item {
        width: 400
        height: 200

        Rectangle {
                id: rect
                x: 100; y: 50;
                width: height / 2; height: parent.height
                anchors.centerIn: parent
                color: "lightblue"
        }
}
```

Wykonanie powyższego kodu spowoduje wyświetlenie okienka z rysunku 1.1. Główny element posiada dziecko będące prostokątem o takiej samej wysokości i szerokości równej połowie jego wysokości. Ponadto prostokąt ten jest wyśrodkowany w swoim rodzicu oraz ma jasnoniebieski kolor.

Jak widać dziecko może odwoływać się do swego rodzica za pomocą słowa *parent*. Z kolei inne elementy mogą się odwoływać do danego elementu za pomocą jego właściwości – id.



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.

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. Dla każdej biblioteki podałem jej zalety 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 programiście szeroki wybór wykresów. Nie wymaga zagłębiania się w swoją wewnętrzną budowę, a tworzenie 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 na komponentach 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.

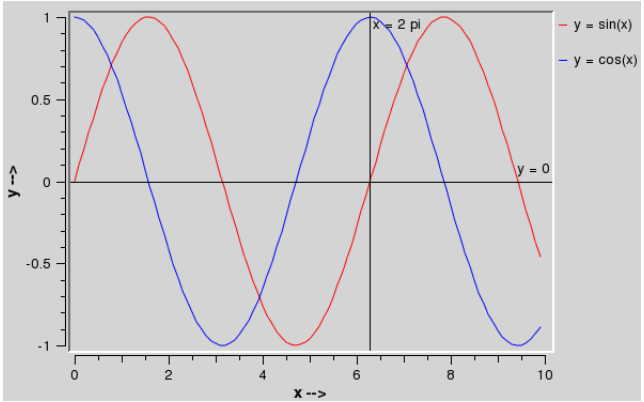
2.2. Rozwiązania darmowe

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 załączony 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, wszystkie 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 czy 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 zwięszkenie 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.

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. Uwzględnienie dostępnych mechanizmów

TODO: delegaty &?

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

3.8.1. Osie

Powinna istnieć możliwość przypisania do osi skali innej niż liniowa. Powinna być możliwość ustawienia pióra służącego do rysowania osi, jej tytułu, gęstości ticków oraz zakresu wartości.

3.8.2. Siatka

Powinna istnieć możliwość określenia grubości i koloru linii oraz ziarnistości samej siatki.

3.8.3. 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.

3.8.4. 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.

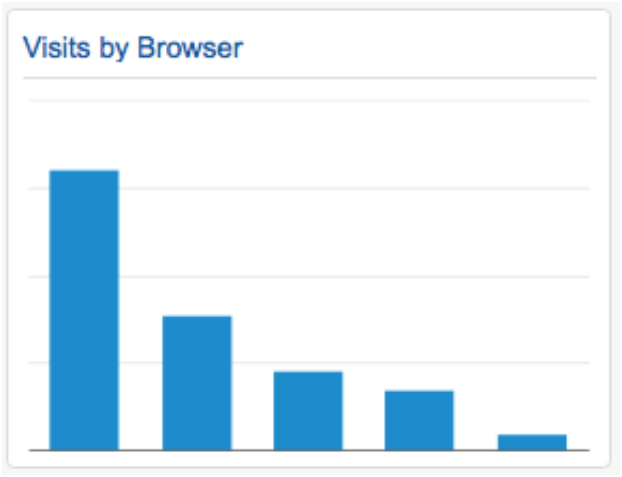
3.10. Wykres słupkowy

Jest to wykres służący do prezentacji danych z wielu serii. Elementem odpowiedzialnym za prezentację pojedynczej próbki jest tu słupek. Wszystkie słupki danej

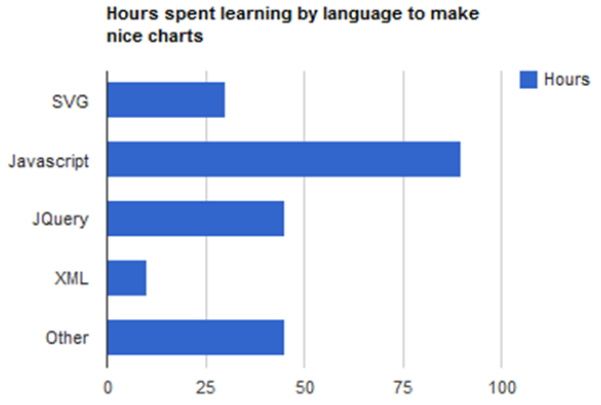
serii mają ten sam kolor. W ogólnym przypadku za pomocą tego wykresu można prezentować wartości z dziedziny liczb rzeczywistych.

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

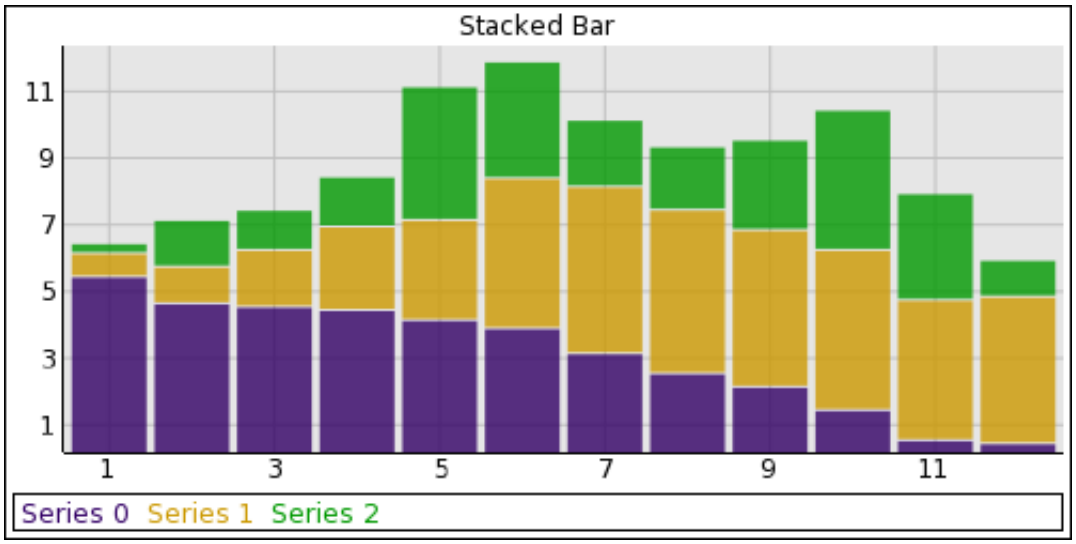


Rysunek 3.2. Poziomy układ słupków

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

Poniżej zostały przedstawione wymagania pozafunkcjonalne stawiane mojej bibliotece.

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 z *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, kwestie 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. Jednakowoż 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.

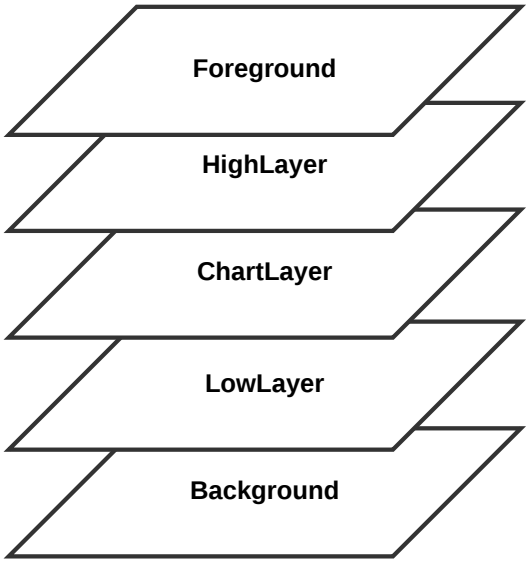
4. Analiza wymagań

5. Projekt

Rozdział ten zawiera szkice rozwiązań dla wymagań z poprzednich rozdziałów.

5.1. Podział na warstwy

Zdecydowałem się podzielić wykres na pięć warstw. Odrysowywanie wykresu będzie się składało z odrysowania 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.

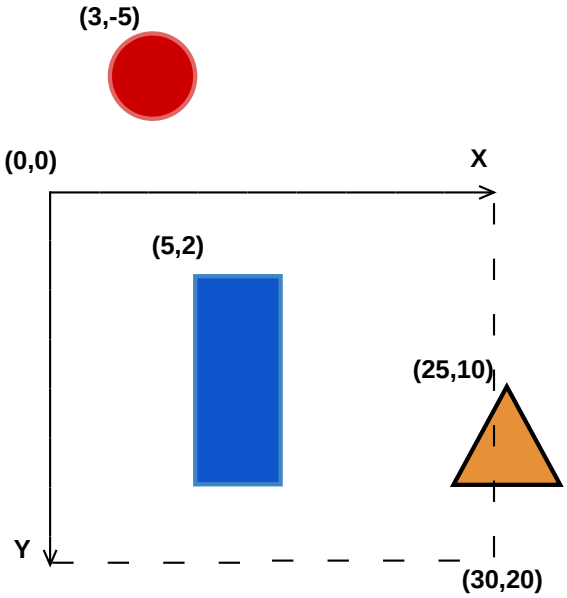
Warstwa wykresu służy odrysowaniu głównej zawartości wykresu. Tym etapem steruje strategia danego wykresu. W ogólnym przypadku programiści nie powinni dodawać własnych elementów do tej warstwy.

Metoda odpowiedzialna za odrysowywanie wykresu powinna być *Metodą Szablonową* [2], czyli niewirtualną, publiczną metodą klasy bazowej, wywołującą kolejne metody odpowiedzialne za odrysowanie pojedynczych warstw. Metody te powinny z kolei być wirtualne i chronione. Metody odpowiedzialne za tło i pierwszy plan

powinny udostępniać domyślną implementację, natomiast pozostałe powinny być czysto wirtualne.

5.2. Lokalny układ współrzędnych

Każdy z wykresów będzie posiadać lokalny, kartezjański układ współrzędnych rzeczywistych. Ponadto granice wykresu będą 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 będzie realizacja 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. Diagramy klas

5.3.1. Top level

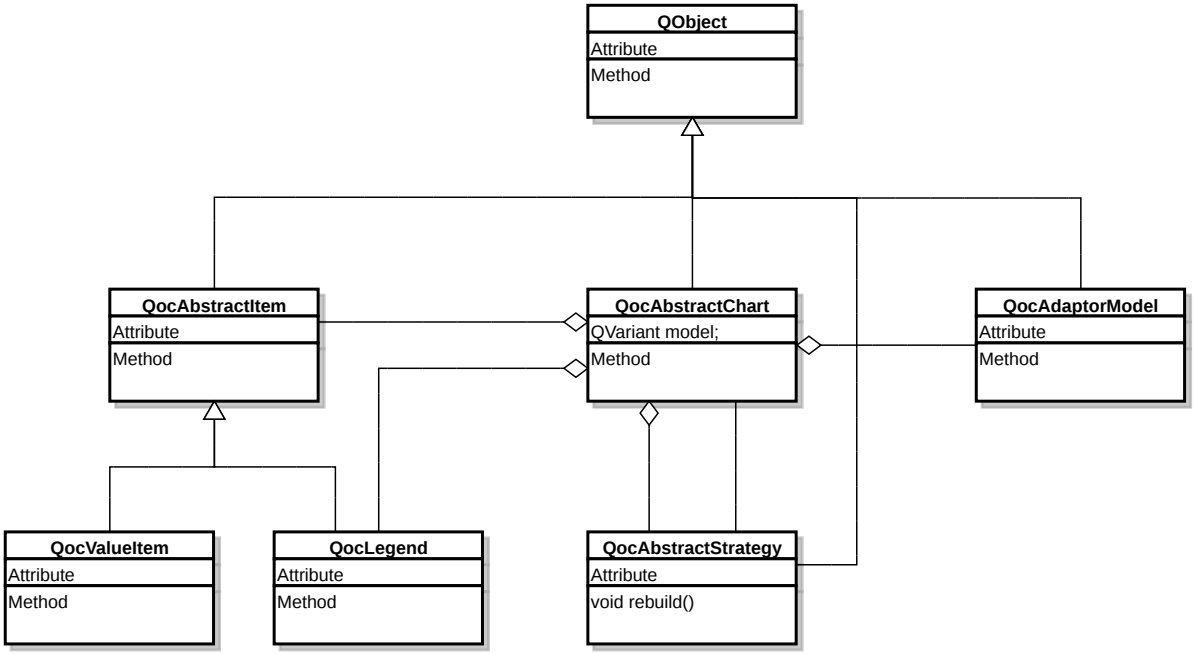
Ogólna koncepcja na strukturę wykresu została przedstawiona na diagramie 5.3.

Klasy pochodne od QocAbstractChart są miejscem łączącym wszystkie inne elementy. To na nich będą 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 –

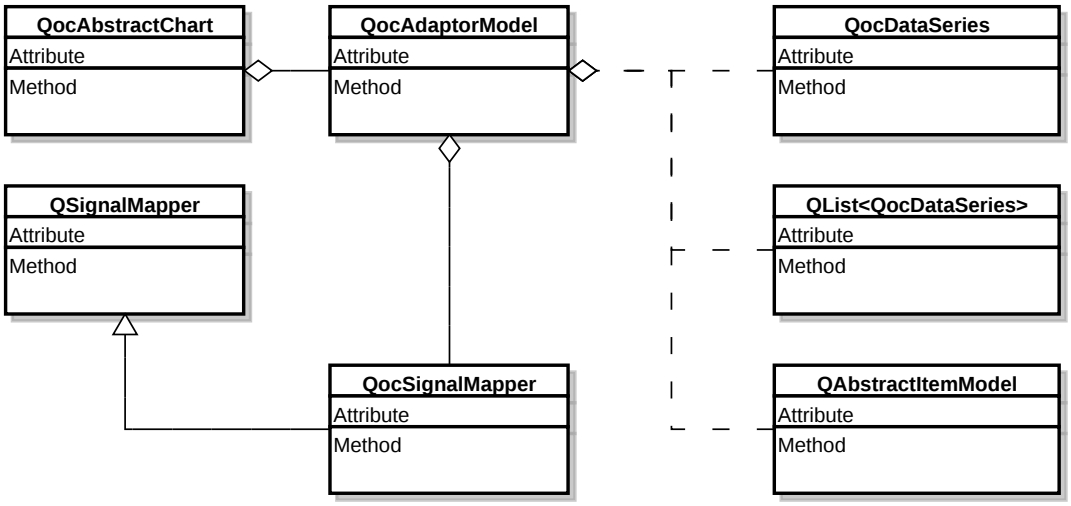
ChartLayer, i ich odrysowywanie. Jest to szczególnie przydatny komponent dla wykresów, które mogą być budowane na różne sposoby, np. wykres słupkowy.



Rysunek 5.3. Diagram top level

5.4. Źródła danych

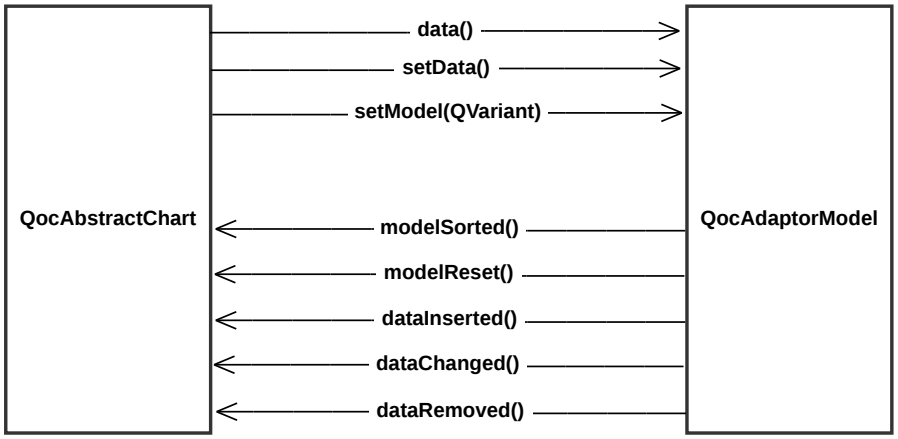
Jako że źródłem danych dla wykresu może być seria danych, zbiór serii albo model, postanowiłem wprowadzić pośrednią klasę, która będzie odpowiedzialna za unifikację komunikacji wykresu ze źródłem danych. Klasa ta będzie *Adapterem obiektowym* [2] – będzie agregowała źródła danych dostarczane jako obiekty *QVariant*. Hierarchię klas związanych z adapterem przedstawiłem na diagramie 5.4



Rysunek 5.4. Adapter

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

Komunikacja w drugą stronę zostanie rozwiązana za pomocą zbioru sygnałów, które również będą niosły informację o id serii. W przypadku zbioru serii, wystarczy uzupełnić ich sygnały o id serii oraz przepropagować na zewnątrz adaptera. Chcę to osiągnąć za pomocą obiektu klasy pochodnej od *QSignalMapper* [1]. W przypadku klas modeli, również będę musiał wykonać podobne mapowanie, podczas którego trzeba będzie sięgnąć do modelu po id 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 uzyskam stan, w którym wykres chcąc pobrać dane z modelu, przekaże do adaptera rolę. Adapter zajmie 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
}
```

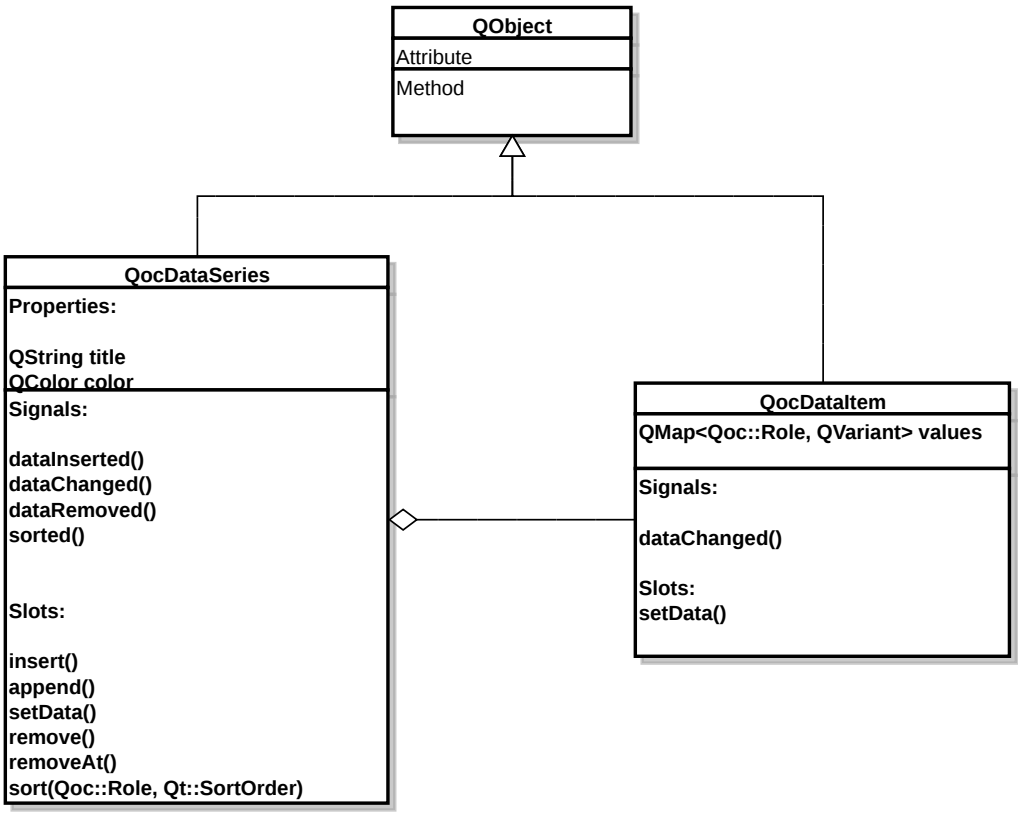
[1] QSignalMapper `http://qt-project.org/doc/qt-5.0/qtcore/qsignalmapper.html`

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 powinna być 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-obj`
[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 powinno się ograniczać 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

5.5.1. Eksport klas C++ do QML

Klasy wszystkich wysokopoziomowych elementów powinny być pochodnymi klasy QObject. Wszelkie ich parametry, które powinny być konfigurowane przez programistów należy włączyć do zbioru właściwości tej klasy. W szczególności należy zadbać o to, 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, powinny zostać opatrzone makrem *Q_INVOKABLE*.

Większość klas powinna być 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 wymagały specjalnych interfejsów. Dla klas związanych z GUI, bazujących na QPainter będzie to QQuickPaintedItem, a dla tych wykorzystujących SceneGraph – QQuickItem.

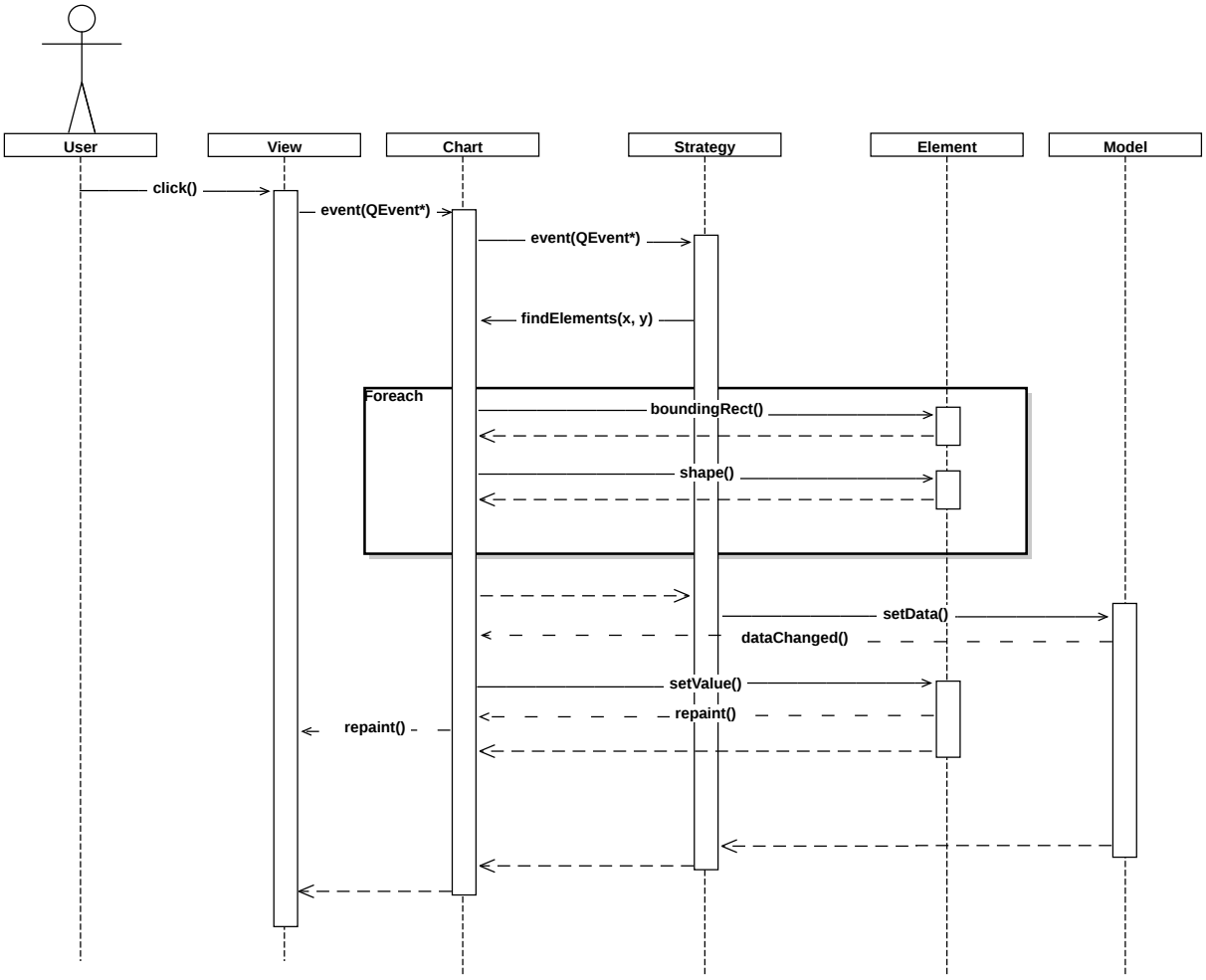
5.5.2. Uproszczony interfejs

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

5.6.1. 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.



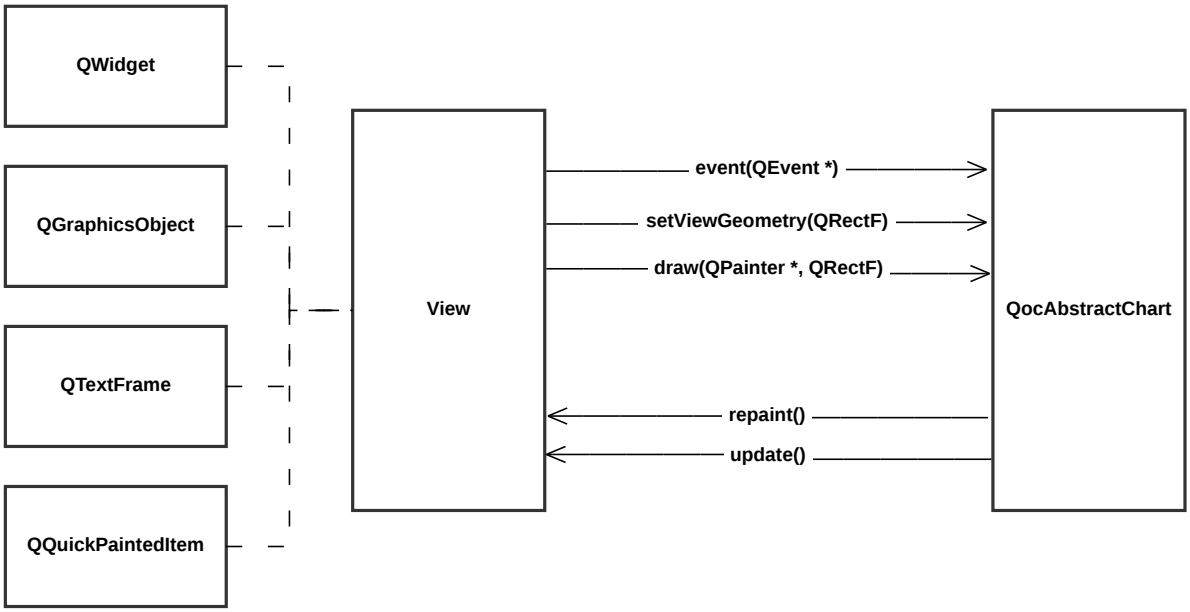
Rysunek 5.7. Interaktywna zmiana zawartości modelu

5.7. 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 5.8 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 powinien posiadać informacji z widokiem jakiej klasy współpracuje. Można to osiągnać za pomocą mechanizmu sygnałów i slotów, który pozwala na luźne wiązanie elementów. Odpowiednie sygnały powinny być emitowane przez wykres zawsze wtedy, gdy wymaga on ponownego odrysowania. Sygnał *repaint()* powinien być 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.



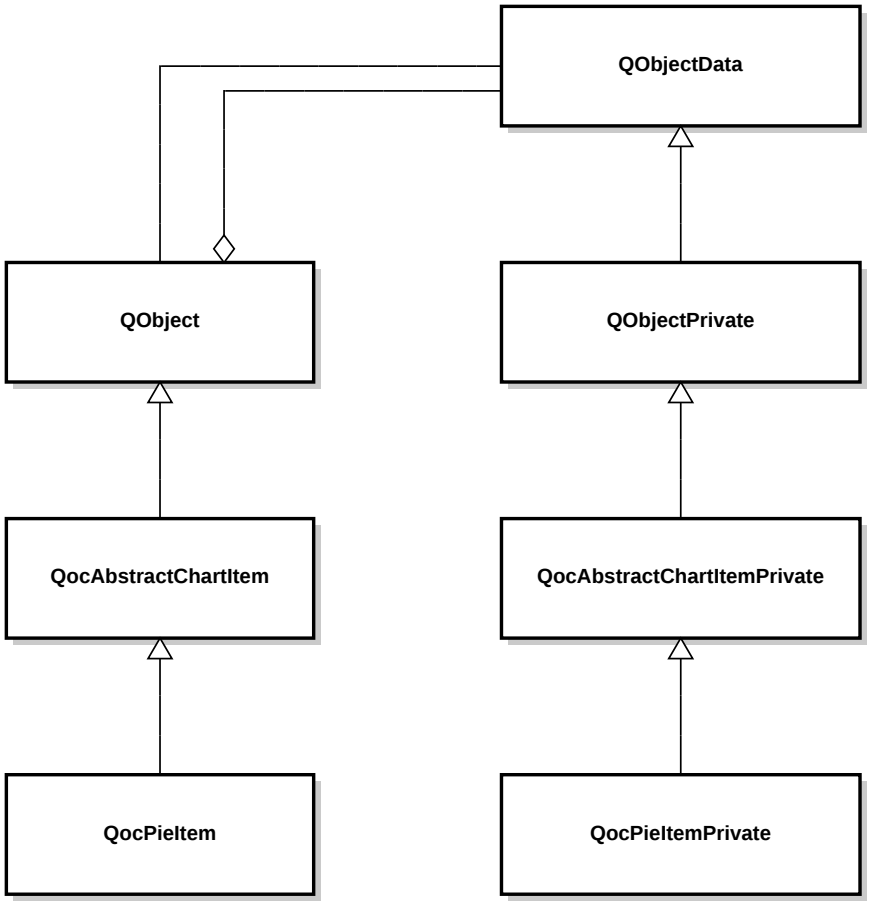
Rysunek 5.8. Widok – Wykres

5.8. 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 [3].

5.8.1. QObject

Klasa QObject została zaprojektowana jako *Most* [2]. 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. 5.9.



Rysunek 5.9. 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”.

5.8.2. Dostęp do uchwytów i ciał

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 [4]

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

Tablica 5.1. Makrodefinicje

Miejsce	Uchwyt	Ciało
Nagłówek	Q_DECLARE_PRIVATE	Q_DECLARE_PUBLIC
Metoda	Q_D	Q_Q

6. Implementacja

Jako, że w ramach pracowni dyplomowej wykonałem sam inżynierski projekt biblioteki, rozdział ten zawiera jedynie informacje o narzędziach, których użyłem przy realizacji projektu oraz opis procesu tworzenia biblioteki współdzielonej za pomocą Qt.

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.

6.1.1. Qt 5

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++.

6.1.2. 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 proste i intuicyjne, a proste GUI można w dużej mierze „wyklikać”.

Do budowania i debugowania Qt Creator wykorzystuje domyślne oprogramowanie danej platformy, np. kompilator gcc i debuger gdb na systemie Linux. Creator posiada graficzny interfejs do debuggera, który w znaczący sposób upraszcza procesz debugowania.

Qt Creator posiada również wtyczki integrujące go z najpopularniejszymi systemami kontroli wersji. Lista wspieranych systemów:
— Bazaar
— CVS
— Git
— Mercurial
— Perforce
— Subversion

6.1.3. 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/`.

6.1.4. 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.

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. Tworzenie biblioteki współdzielonej

Biblioteka współdzielona to rodzaj biblioteki dynamicznej, czyli biblioteki łączonej z programem dopiero w trakcie jego uruchamiania. Przy pierwszym uruchomieniu programu korzystającego z danej biblioteki współdzielonej, biblioteka ta zostaje załadowana w całości do pamięci. Od tej pory wszystkie programy, będą mogły korzystać z tej biblioteki bez ponownego jej ładowania. W systemach Unixowych biblioteki dynamiczne mają rozszerzenie .so, a w systemach Windows .dll.

Qt udostępnia nam stosunkowo prosty sposób na tworzenie bibliotek dynamicznych.

6.2.1. Symbole

Wszelkie funkcje, zmienne i klasy zawarte w bibliotece, które są przeznaczone do użytku dla klientów biblioteki są nazywane publicznymi symbolami i muszą zostać wyeksportowane, czyli upublicznione podczas kompilacji biblioteki. Zazwyczaj domyślne zachowanie kompilatorów to ukrywanie wszystkich symboli. Aby stały się one dostępne dla klientów biblioteki, trzeba to otwarcie zasygnalizować podczas kompilacji. Natomiast na niektórych platformach osobnych instrukcji wymaga ukrycie symboli.

6.2.2. Eksport – Import

Rozwiązaniem obu problemów są dwa makra dostarczane przez Qt, które muszą zostać dodane do deklaracji symboli:
— Q_DECL_EXPORT – przy kompilacji dynamicznej biblioteki
— Q_DECL_IMPORT – przy kompilacji klienta korzystającego z biblioteki

[1] http://code.google.com/intl/pl/
[2] Kubuntu `http://pl.wikipedia.org/wiki/Kubuntu`

6.2.3. Uniwersalne makro

Teraz trzeba się upewnić, że odpowiednie makro zostanie wykorzystane w odpowiednim momencie. Typowym rozwiązaniem jest dodanie specjalnego pliku nagłówkowego, np. *foolib_global.h*. Plik ten musi zawierać kod preprocesora realizujący następującą sztuczkę:

Wydruk 6.1. Warunkowa kompilacja

```
#include <QtCore/QtGlobal>

#if defined(FOOLIB_LIBRARY)
#  define FOOLIB_API Q_DECL_EXPORT
#else
#  define FOOLIB_API Q_DECL_IMPORT
#endif
```

Jak widać, w zależności od tego czy istnieje makro *FOOLIB_LIBRARY*, makro *FOOLIB_API* posłuży do eksportowania albo importowania symboli biblioteki *FooLib*. Makro *FOOLIB_LIBRARY* zostanie zdefiniowane tylko i wyłącznie w pliku projektu biblioteki *FooLib*. Tylko wtedy będziemy mieli pewność, że nasza biblioteka będzie poprawnie linkowana. W pliku .pro należy dodać instrukcję: *DEFINES += FOOLIB_LIBRARY*.

6.2.4. Wykorzystanie makra

We wszystkich plikach nagłówkowych biblioteki, deklaracje symboli muszą zostać uzupełnione o nasze makro:

Wydruk 6.2. Eksport symboli

```
#include "foolib_global.h"

FOOLIB_API void foo();
class FOOLIB_API Foo...
```

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 "CONFIG += qtestlib"
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. Każdy z tych testów powinien być uruchamiany podczas testów regresji.

7.2.1. Operacje na serii danych

Test będzie polegał na dodawaniu, usuwaniu i modyfikowaniu danych w serii. Po każdej operacji zawartość serii będzie porównywana z oczekiwaną zawartością.

7.2.2. Dostarczanie zdarzeń

Planuję przesyłać do wykresu zdarzenia, które powinny powodować zaznaczenie, bądź odznaczenie elementu wykresu. Przesłane w zdarzeniu współrzędne powinny jednoznacznie wskazać, który z elementów powinien zostać zaznaczony. Należy przetestować również wysyłanie zdarzeń nie trafiających w żaden element.

7.2.3. Modyfikacja modelu z poziomu wykresu

W tym teście preparowane będą zdarzenia imitujące proces mający na celu zmianę danych zawartych w modelu poprzez modyfikację elementów wykresu.

7.2.4. Kolejne przypadki testowe...

8. Podsumowanie

8.1. Wnioski

8.2. Kierunki rozwoju

Bibliografia

[1] Jasmin Blanchette. The little manual of api design. 2008.

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

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

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