

Politecnico di Milano
Facoltà di Ingegneria dell'Informazione



Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Evaluation of Qt as GUI Framework
for Accelerator Controls

Relatore: Elisabetta DI NITTO

Tesi di Laurea di: Sara ZANZOTTERA matr. 880407

Anno Accademico 2017-2018

Acknowledgments

I would like to express my deepest gratitude to Vito Baggiolini, my main supervisor, for his patience, encouragement and constant support, especially when the difficulty of some tasks overwhelmed me. Most of what I achieved would not be possible without his help. I owe him all that I learned at CERN and much more.

I would also like to mention Eric Roux, Felix Ehm, Mathieu Gabriel, Maciej Peryt, Andres Rodriguez Perez, Nuno Mendes and all my colleagues that sooner or later joined the one-man's team I were, and helped me along the way. Working with you was enlightening, under every perspective.

Special thanks to Bálint Hegyi for his invaluable help on all the side tasks, and for his interest in my work.

Many thanks also to Elisabetta Di Nitto, my university supervisor, for her careful reviews of my work and her helpful remarks.

Nothing of this, however, would have been possible without the help of my friends and relatives. A special mention to Gianpaolo Branca for his generous and selfless support, to my sister Alice, to Simone Mosciatti, to Ivan Vigorito, to all my colleagues of the section and of the group, who made a very supporting, informal and friendly environment on the workplace.

And, on purpose last, I should mention my delegate supervisor, *főnökök* Zsolt, whose exceptional ability to change any little thing into a challenge will not be forgotten.

Geneva, November 29, 2018

Sara Zanzottera

Sommario

Il complesso degli acceleratori del CERN è una successione di macchine in grado di accelerare diversi tipi di particelle a livelli di energia crescenti. Ognuna di esse aumenta drasticamente l'energia del fascio di particelle proveniente dalla precedente e lo invia alla successiva, che a sua volta accelera ulteriormente il fascio. Nel Large Hadron Collider, l'ultimo elemento della catena, il fascio di particelle viene accelerato all'energia record di 6.5 TeV per fascio.

Uno strumento simile non potrebbe mai operare senza un efficiente sistema di controllo, ma il monitoraggio della macchina non è un problema di minore rilevanza e non deve essere sottovalutato. La strumentazione di controllo del CERN invia costantemente dati aggregati e dati grezzi ai livelli superiori del sistema di controllo, in genere un'interfaccia grafica, che è incaricata di mostrare i suddetti dati agli operatori della Sala di Controllo in maniera efficace e con dei requisiti piuttosto stretti sulla performance. Questi sistemi sono critici per il CERN, in quanto permettono agli operatori di monitorare gli acceleratori e regolarli in maniera efficace, tale da generare la maggior quantità di dati scientifici possibile senza danneggiare le macchine e la strumentazione.

Anche per questa ragione, i requisiti posti sulle interfacce grafiche e sul sistema di controllo vengono costantemente aggiornati per restare in linea con i cambiamenti che avvengono nella strumentazione, mentre le tecnologie per lo sviluppo delle interfacce grafiche evolvono in direzioni diverse e spesso divergenti. Di conseguenza, si è reso necessario investire tempo e risorse nella ricerca di un valido compromesso tra le caratteristiche dei moderni GUI framework e le necessità degli operatori.

La tesi è incentrata sul mio contributo agli sforzi del dipartimento di mantenere le interfacce grafiche del sistema di controllo aggiornate, efficaci e veloci, utilizzando tecnologie moderne. A questo scopo ho lavorato nel Beams Department, Controls group, Applications section in un team di quattro ingegneri con

una lunga esperienza nello sviluppo di interfacce grafiche, e sono stata messa in contatto con la comunità degli sviluppatori del CERN e con gli operatori della Sala di Controllo.

Il Contesto

I livelli superiori del sistema di controllo degli acceleratori del CERN sono scritti principalmente in Java, con circa 400 applicazioni server e 600 diverse interfacce grafiche. In totale si tratta di circa 10 milioni di linee di codice e più di 1000 file .jar, creati da una comunità di circa 120 sviluppatori Java. Circa metà di questi produce interfacce grafiche e hanno diversi background: non si tratta solo di ingegneri del software o programmatori, ma anche fisici delle particelle, esperti di fisica degli acceleratori, operatori, esperti hardware e persino periti tecnici.

Java è da molto tempo uno standard affermato nello sviluppo di software di livello industriale, ed è ancora una scelta molto solida per quanto riguarda lo sviluppo di un sistema informatico vasto come il sistema di controllo degli acceleratori del CERN. Nonostante ciò, Java offre una scelta piuttosto limitata per quanto riguarda le tecnologie di sviluppo delle interfacce grafiche: le opzioni si limitano all'ormai obsoleto AWT, il suo successore Swing, e al più moderno JavaFX, che è stato sviluppato con l'obiettivo di sostituire Swing come principale GUI framework per le applicazioni Java.

Al di là di queste tre opzioni, alcuni vendors hanno sviluppato framework minori per lo sviluppo delle interfacce grafiche, come ad esempio SWT (Standard Widget Toolkit, sviluppato dalla Eclipse Foundation), o GWT (Google Web Toolkit), i binding su Java per OpenGL (JOGL) o altri bindings per librerie grafiche 3D come Java3D^[1].

Il Problema

La maggior parte delle interfacce grafiche del sistema di controllo sono scritte in Swing, mentre molte delle più recenti sono sviluppate in JavaFX. Nonostante gli sforzi compiuti dalla mia sezione per fornire supporto nello sviluppo di applicazioni in JavaFX e promuoverne l'utilizzo nella comunità degli svilup-

patori, si è notato che al di fuori del CERN l'interesse verso questo framework è andato decrescendo, in quanto la maggior parte dell'interesse si muove ogni giorno di più verso le tecnologie Web.

In aggiunta a tutto ciò Oracle, che finora ha mantenuto e sviluppato JavaFX, ha recentemente annunciato che terminerà il supporto per JavaFX nel 2022^[3] e che rilascerà il progetto come open source, in modo che la comunità possa continuare a utilizzarlo e svilupparlo liberamente^[6].

Una simile dichiarazione, assieme al generale calo di interesse verso le tecnologie desktop in generale, ha suscitato parecchie preoccupazioni riguardo al futuro di JavaFX. Si è quindi reso interessante esplorare le alternative esistenti, per capire quale tecnologia potrebbe diventare mainstream per lo sviluppo di GUI desktop nel prossimo futuro. In questo modo il dipartimento potrebbe orientare fin da subito i propri sforzi verso un framework più promettente.

Un aspetto positivo, se così si può definire, è il fatto che il generale calo di interesse nel campo delle GUI desktop ha accelerato l'obsolescenza dei framework più deboli, mettendo in evidenza i pochi framework più solidi. Da questo punto di vista, Qt è sicuramente una delle tecnologie che si è distinta immediatamente.

Qt^[7] è probabilmente il più utilizzato framework per GUI, fatta eccezione per le tecnologie Web. Ha un buon numero di *success stories*, per esempio KDE^[8;9] ed è mantenuto da un'azienda il cui business model ruota interamente attorno ad esso^[36]. In generale, Qt pare una tecnologia promettente da qui al prossimo futuro.

D'altra parte Qt è un framework estremamente complesso e vasto, che può essere utilizzato in molti modi diversi. Ha anche un buon numero di problemi rispetto a JavaFX, ognuno dei quali va analizzato nel dettaglio prima di poter promuovere il framework come valida alternativa.

L'Obiettivo

L'obiettivo di questa tesi consiste nell'effettuare un'analisi esplorativa di Qt, per verificare se e come è in grado di interoperare efficacemente con il backend Java che genera e pubblica i dati.

Gli obiettivi principali includono:

-
1. Identificare tutti i possibili approcci e binding su altri linguaggi offerti da Qt che sono adatti a un'ispezione più approfondita, basandosi sulla loro popolarità nella comunità di Qt, le caratteristiche offerte e la probabilità che si tratti di una soluzione adatta ai nostri requisiti.
 2. Familiarizzare con ciascuno degli approcci individuati, la loro struttura e le loro capacità. In molti casi ciò implica imparare il linguaggio o i linguaggi coinvolti (C++, JNI, JavaScript, TypeScript, QML, Python, etc...) e a utilizzare i tool necessari per lo sviluppo.
 3. Valutare ciascuno degli approcci dal punto di vista della disponibilità degli widgets necessari per i sistemi di controllo (menu, grafici, inputs, popups, tooltips, worker threads, etc...) rispetto ai requisiti posti dal CERN.
 4. Scrivere dei report dettagliati riguardo ai risultato della valutazione e offrire al section leader tutte le informazioni necessarie per prendere decisioni informate in merito alla strategia per lo sviluppo delle GUI nei prossimi anni.

Struttura della Tesi

Prima di dare il via alla valutazione è stata eseguita un'analisi preliminare delle tecnologie disponibili come alternative a JavaFX e, come già illustrato sopra, il primo candidato per l'analisi è risultato essere Qt.

Qt è stato selezionato soprattutto per la vastità di opzioni che offre agli utenti, la solidità del framework, e per la presenza di diversi binding con altri linguaggi^[7]. Ciò significa che Qt offre diversi approcci allo sviluppo delle interfacce, ognuno dei quali è stato valutato separatamente.

Contemporaneamente a questa analisi, è stata eseguita anche una rapida valutazione di JavaFX, in modo da offrire un riferimento per la valutazione di Qt.

Date queste premesse, la tesi è organizzata come segue:

- Il Capitolo 1 descrive il contesto della ricerca, il problema posto, le motivazioni, il metodo e i risultati attesi.

-
- Il Capitolo 2 presenta una descrizione più dettagliata del metodo di valutazione. Dopo una breve introduzione allo stato dell'arte, l'applicazione benchmark è descritta nel dettaglio sia da un punto di vista grafico che da un punto di vista funzionale. Infine viene presentata la lista degli approcci selezionati, assieme a una breve spiegazione delle ragioni della selezione.
 - Il Capitolo 3 illustra i risultati dell'analisi di JavaFX, in modo da offrire un solido punto di riferimento per la successiva valutazione di Qt.
 - Il Capitolo 4 è un'introduzione a Qt in generale e contiene un'introduzione alle sue caratteristiche principali. Lo scopo del capitolo è di aiutare il lettore a capire meglio le scelte compiute in seguito durante la valutazione dei vari approcci.
 - Il Capitolo 5 presenta i risultati della valutazione di QtJambi, un binding su Java per Qt 4, e il nostro tentativo di farlo funzionare con l'ultima versione di Qt, la versione 5.
 - Il Capitolo 6 descrive la valutazione del metodo più standard per lo sviluppo di GUI in Qt: le applicazioni Qt Widgets, sviluppate in C++.
 - Il Capitolo 7 illustra i vari approcci considerati durante lo sviluppo dell'applicazione benchmark in QtQuick, una tecnica di sviluppo di GUI in Qt che non utilizza C++, ma QML e JavaScript.
 - Il Capitolo 8 presenta i risultati della valutazione di PyQt5, il binding di Qt su Python più stabile e supportato. Con PyQt5 sono state testate due applicazioni, una utilizzando lo stile Qt Widgets e una in QtQuick.
 - Il Capitolo 9 riassume i risultati di tutte le diverse analisi e trae alcune conclusioni sull'intero processo di valutazione.

Table of Contents

- 1 Introduction** **1**
 - 1.1 The Context 2
 - 1.2 The Problem 3
 - 1.3 The Task 5
 - 1.4 Outline of the Thesis 5

- 2 Evaluation Outline** **9**
 - 2.1 State of the Art 9
 - 2.2 Evaluation Strategy 11
 - 2.3 The Benchmark Application 13
 - 2.3.1 The Interface 14
 - 2.3.2 The Data Source 16
 - 2.3.3 Charting performance assessment strategy 16
 - 2.4 Selected Technologies 18
 - 2.4.1 Web Technologies 19

- 3 Current State: JavaFX** **23**
 - 3.1 Installation and setup 23
 - 3.2 Development process 24
 - 3.3 The application 25
 - 3.3.1 Charting 27
 - 3.4 Tooling 28
 - 3.5 Documentation 29
 - 3.6 Outcomes 29

- 4 Qt: An Overview** **31**
 - 4.1 Framework architecture 32

TABLE OF CONTENTS

4.1.1	QtQuick <i>versus</i> QtWidgets	32
4.1.2	Signals and slots	34
4.1.3	The Meta Object System	34
4.2	Qt as CERN's GUI Framework	35
4.2.1	A C++ Framework	35
4.2.2	Qt Bindings	35
5	Qt over Java: QtJambi	37
5.1	Installation and setup	37
5.1.1	The QtJambi Generator	38
5.1.2	Actual installation process	39
5.2	Development process	40
5.3	The application	40
5.4	Tooling	42
5.5	Documentation	42
5.6	Outcomes	42
6	Qt Widgets	45
6.1	Installation and setup	45
6.2	Development process	47
6.2.1	Qt's Meta Object System	48
6.2.2	GUI Design	49
6.2.3	Build Process	50
6.3	The application	51
6.4	Tooling	51
6.5	Documentation	52
6.6	Outcomes	52
7	Qt Quick: QML & JavaScript	55
7.1	Installation and setup	56
7.2	Development process	56
7.2.1	Layout System	57
7.2.2	QML Components	58
7.2.3	JavaScript Code	58
7.2.4	JavaScript Host Environment	59
7.2.5	Charting	60

TABLE OF CONTENTS

7.3	The application	61
7.4	Tooling	64
7.5	Documentation	64
7.6	A Side Attempt: TypeScript	65
7.6.1	Node.js Modules	66
7.6.2	Tooling	66
7.6.3	Evaluation Outcomes	67
7.7	Outcomes	68
8	Qt over Python: PyQt5	69
8.1	Installation and setup	70
8.2	Development process	70
8.2.1	Code Comparison	70
8.2.2	QtQuick	72
8.2.3	QtWidgets	73
8.3	The application	73
8.3.1	QtQuick	73
8.3.2	QtWidgets	77
8.4	Tooling	79
8.4.1	QtQuick	79
8.4.2	QtWidgets	80
8.5	Documentation	80
8.6	Outcomes	81
9	Conclusions	83
	Appendices	85
A	Acronyms Definition	87
A.1	Definitions	87
B	A JavaX Hello World Application	89
B.1	First Approach: Pure Java	89
B.2	Hello World, MVC enforced	91
C	QtJambi Deployment Layout	93

TABLE OF CONTENTS

D Final Comparison Table Justifications	97
D.1 JavaFX	98
D.2 QtJambi (for Qt5)	99
D.3 QtQuick (in pure JS)	99
D.4 QtWidgets	100
D.5 PyQt5 QtQuick	100
D.6 PyQt5 QtWidgets	101
Bibliography	103

Chapter 1

Introduction

The accelerator complex at CERN is a succession of machines that accelerate particles to increasingly higher energies. Each machine boosts the energy of a beam of particles, before injecting the beam into the next machine in the sequence. In the Large Hadron Collider (LHC) – the last element in this chain – particle beams are accelerated up to the record energy of 6.5 TeV per beam.

Such a machine could not operate without an effective and performant control system, but the need for an effective visualization system must not be underestimated. CERN's accelerator controls technologies constantly feed aggregated and status data to the uppermost layer of the control system, the user interface, that is in charge of displaying them to the operators of the Control Room, with strict requirements on performance. These systems are mission critical for CERN, as they give the operators the possibility to monitor the accelerators and run them effectively to produce the largest possible amount of scientific data, while operating the machine safely.

In addition, the requirements of the Control Room operators are constantly changing, as the underlying systems improve, while user interface technologies evolve in directions that not always match CERN's requirements on the matter. Therefore constant research must be poured into finding the best compromise between modern visualization framework's features and the accelerator control's needs.

In this report I am going to describe my personal contribution to the organization's effort to keep their user interface stack up to date, performant and effective, using modern frameworks and technologies. As such, I am employed by the Beams department, Controls groups, Applications section. I am part

Introduction

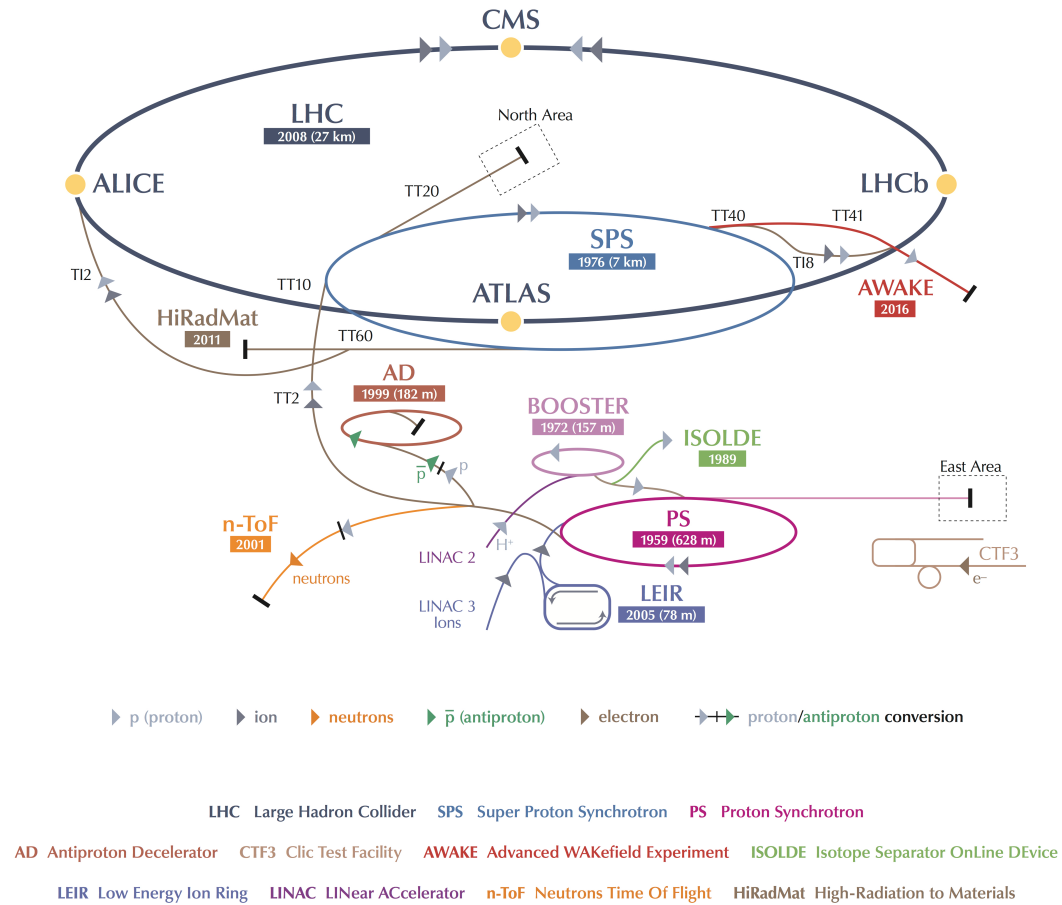


Figure 1.1: CERN's Accelerators Complex^[2]

of a team of four software engineers with strong background in user interfaces development, and I have been put in direct contact with the developer community of CERN and the operators in the Control Room.

1.1 The Context

The higher levels of CERN's accelerator control system are written mostly in Java, with around 400 server applications and 600 different GUIs. So far, they amount to roughly 10 million lines of source code and more than 1000 .jar files, created by a community of around 120 Java developers. Around half of those developers write graphic user interfaces (GUIs) and they have different backgrounds: not only software engineers, but also accelerator physicists, operators, hardware experts and service technicians.

Java has long been an industry standards for production-level software, and it is still a very solid choice as base programming language for such a large software base as CERN's accelerator's control systems is. Nonetheless, it offers a limited choice regarding GUI development frameworks: the now obsolete AWT framework, its successor Swing, and the latest JavaFX, that was meant to replace Swing as main GUI development framework for Java applications.

Other than these libraries, some vendors have also provided graphics APIs that work with Java, such as Eclipse's Standard Widget Toolkit (SWT), Google Web Toolkit (GWT), 3D Graphics API such as Java bindings for OpenGL (JOGL) and Java3D^[1].

1.2 The Problem

Most of the GUIs of the accelerator's controls are written in Java Swing, and some of the most recent ones in JavaFX. However, such technologies has been decreasingly popular among the community, that is now focusing more and more on web-based GUI technologies leaving desktop-based GUIs behind.

In addition to these concerns, Oracle, the maintainer of JavaFX, recently stated that it will drop support to the project by 2022^[3] and leave the maintenance effort to the open source community^[6].

Such a statement, combined with the adverse environment surrounding desktop GUI frameworks, raised serious concerns about the future of JavaFX. Therefore it became interesting to explore possible alternatives to JavaFX and to investigate on new technologies that may become the next mainstream framework for desktop GUI development, in order to orient the development efforts in the right direction as soon as possible.

On the other hand, this general decline of interest in the entire area of desktop GUI development quickly let the most robust widget frameworks to stand out, as soon as their competitors died out. In this perspective, Qt showed clearly as one of the most interesting alternatives.

Qt^[7] is likely the most popular desktop GUI framework available, let alone web-based technologies. It has a number of success stories, namely the Linux KDE GUI^[8;9], and is maintained by a company whose business focuses entirely on it^[36]. In general, it looks popular and promising for the foreseeable future.

Introduction

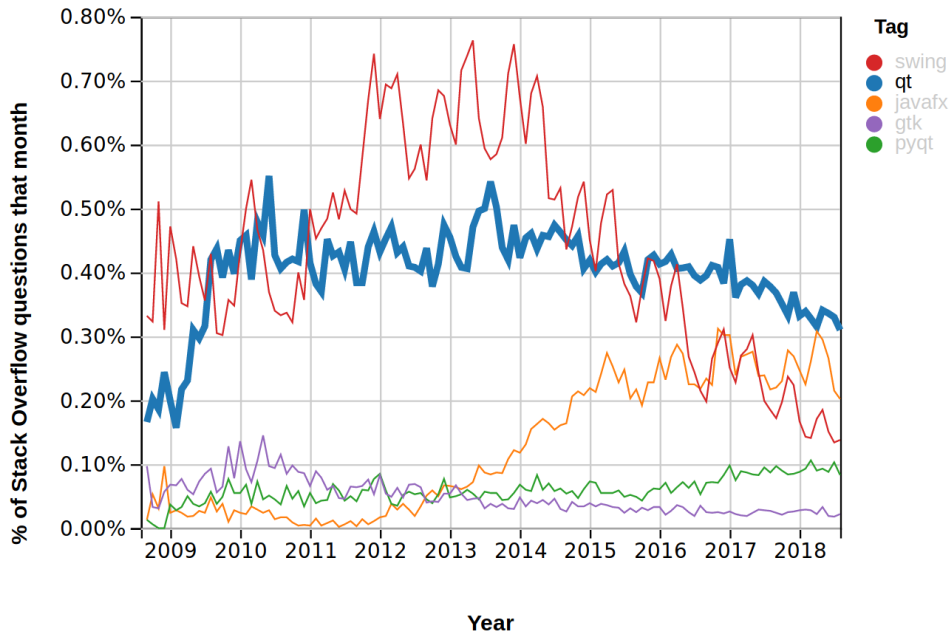


Figure 1.2: StackOverflow Trends^[4] over a 10 years period. While most desktop GUI technologies seems to be fading out, Qt is the one whose descending trend seems to be the slowest. Note also PyQt stability, even if clearly less popular. The evident growth of JavaFX is also on hold and is not likely to restart.

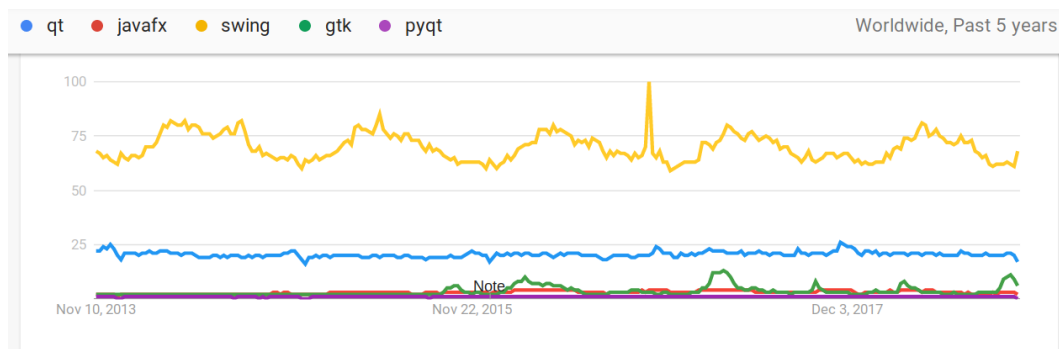


Figure 1.3: Google Trends^[5] over a 5 years period. With respect to the StackOverflow plot above, Swing seems to be much more popular than any other widget toolkit, Qt being second with a very stable profile. JavaFX, instead, is quite less relevant. Is interesting to notice that this is a plot of Interest over Time, defined as "search interest relative to the highest point on the chart for the given region and time."

Nonetheless, Qt is an extremely wide and complex framework which can be used in different ways, and it may have a number of drawbacks compared to JavaFX, each of whose has to be explored in detail before committing to it.

1.3 The Task

My tasks are focused on an exploratory approach of Qt to find out if and how it can inter-operate smoothly with the Java backend providing the data. The main goals include:

1. Identify all the suitable modules and bindings of Qt to be further evaluated, basing on their popularity, their stated capabilities, their apparent suitability to our needs.
2. Become familiar with the different modules that are found relevant, their architecture and their capabilities. In most cases, this step included learning the relevant programming language of the framework itself and how to use the related tools.
3. Evaluate, for each of the modules, all relevant components of a typical accelerator controls GUI (menus, charts, input elements, pop-ups, tooltips, background tasks, actions, an event bus, ...), with special care to CERN's specific needs.
4. Write a report that compares the different possibilities according to the criteria described below, and present the findings to the Beams Department's developer community.

1.4 Outline of the Thesis

Qt has been chosen for the evaluation especially for its comprehensiveness, solidity, and for the presence of bindings to many programming languages^[7]. As I am going to detail in following sections, Qt provides several different approaches for GUI development, most of which has been evaluated separately.

In parallel, a quick re-evaluation of the Java GUI frameworks already in use (which is JavaFX) has been carried on, in order to provide a reference to evaluate the other frameworks with.

Introduction

Basing on these premises, the thesis has been organized as follows:

- Chapter 1 describes the overall context along with the problem being faced by the organization, the rationale behind the research, its main criteria and the expected outcomes.
- Chapter 2 presents a more in-depth description of the evaluation strategy. After a brief introduction to the state of the art in software toolkits evaluation, the benchmark application is described both from an UI perspective and from a functional point of view. Finally a list of the candidate modules and bindings is presented, along with the selection rationale.
- Chapter 3 details the outcomes of the re-evaluation of JavaFX as GUI toolkit, in order to provide a good reference for the subsequent evaluation of Qt.
- Chapter 4 is meant to be an introduction to Qt in general and an overview of some of its most prominent features, to help the reader understanding the main reasoning behind some of the choices we made along the entire evaluation.
- Chapter 5 presents the outcomes of the evaluation of QtJambi, a Java binding for Qt 4, and our attempts to adapt the software to work with the latest version of Qt, version 5.
- Chapter 6 describes the work done on the most standard and basic way of developing Qt GUIs in C++: Qt Widgets-style applications.
- Chapter 7 details the various efforts and approaches we took during the evaluation of QtQuick-style development of Qt application: pure QML and JavaScript approach, the TypeScript approach, etc...
- Chapter 8 presents the results of the evaluation of PyQt5, the most stable, healthy and solid binding of Qt to another programming language now available. We tested it both for Qt Widgets and for QtQuick style applications.

1.4 Outline of the Thesis

- Chapter 9 sums up the outcomes of all the evaluations performed and draws a conclusive picture of the possibility to adopt Qt as main GUI framework for CERN's control systems.

Chapter 2

Evaluation Outline

GUI toolkits are some of the most complex and varied software families in existence today, and also some of the oldest. Depending on their age, their maturity level, their aim and their history, their complexity level and approach toward interface development varies wildly. Even having the analysis limited to the sole Qt framework, carrying on a coherent evaluation of all the different approaches available for GUI design is a very challenging task, that must be performed with special care and discipline.

In this chapter I am going to detail the strategy we followed for the evaluation of Qt, while the results will be outlined, separately for each approach tested, in the following chapters.

2.1 State of the Art

Careful evaluation of a company's software tools is a very common task, but is often overlooked.

Even if it is a problem as old as software development itself, the ratio between the investment in an evaluation and the benefits that come from it is perceived to be very low and therefore ignored, often leading to costly mistakes in terms of money and time.

As A. Powell effectively describes in one of his articles^[12], the reasons behind this lack of interest in the matter are very diverse and sometimes even legit, but the consequences they lead to can be unpredictably expensive for developers and managers alike. Even if the paper dates back to the more than

Evaluation Outline

twenty years ago, his description of the situation is very actual and still reflects the working environment of many companies.

The article also highlights the lack of a common and solid methodology for software evaluation tools and the unavoidable consequences of this situation on companies. Indeed, according to the authors, a solid evaluation framework should be able to capture at the same time the "hard" features of a software tool and its "soft" features, like usability with the rest of the company's toolchain, the emplacement of the new tool (that is, the process of replacing the old tool with the new one), and so on. If some of these soft features are missing, most of the tool's alleged benefits may fail to materialize and, in some cases, even become a burden.

This is especially true nowadays in some context where software tool adoption seems to be driven "by fashion" instead of by actual need, as it seems to happen for example in the context of JavaScript web frameworks^[13]

Nowadays the situation is clearly improving. In an investigation from BI Survey^[16] more than 63% of the companies queried perform some kind of competitive, formal software tool evaluation before adopting it, while 17% carries on single product evaluation and 20% no evaluation at all.

Another interesting outcome is that, when comparing the market performance of these companies, is quite evident how a careful software evaluation correlates with better market performance. In addition, they claim that in their sample, business intelligence users are rarely able to calculate the hard return of their evaluations, but they notice qualitative improvements in the speed and accuracy of the decisions taken.

Structured approaches for generic software evaluation can be easily found online, and some companies even provide the evaluation as a service. One example is the Software Sustainability Institute, which provides both a consultancy service on the topic^[14], and a few free guides on how to perform the evaluation effectively^[15].

Their guide covers the evaluation process from beginning to end, and in particular underlines the following points:

- The evaluation should be two-fold: on one side, it must capture the quantitative features of the software under evaluation, its performance, its ability to fulfill the requirements in terms of sustainability, main-

tainability, and usability. On the other hand, it should also assess the qualitative features, defined as "a pragmatic evaluation of usability of the software in the form of a reproducible record of experiences". This gives a developer a practical insight into how the software is approached and any potential technical barriers that prevent adoption.

- The targets of our evaluation. The evaluation is done for someone, and the targets may be multiple and different. For example we may be doing an evaluation for software users, for user-developers (like in case of a library or a framework), for developers (that may want to modify the library itself), for the management (which may be more interested in the soft characteristics than in the hard features) and so on.
- How to properly write a report with the outcomes of the evaluation depending on the target audience.

Plenty of other software evaluation methods, matrices, tables and criteria can be found online with a quick search^[17]. However, it is pretty evident how the procedure is not normalized or, at least, a common and clear evaluation framework is not yet well established.

2.2 Evaluation Strategy

To carry on our evaluation of Qt with respect to JavaFX, we did our best to keep into consideration the main guidelines and advises found. The results are explained in the following section, with the hope of clarifying the rationale behind our choices during the actual evaluation phase.

In order to achieve meaningful results, the evaluation strategy has been focusing on two main dimensions of the framework under evaluation:

1. The *performance* of the toolkit itself under specific kinds of load.

Our section is in charge of providing solutions for two main use cases: advanced controls widgets (like knobs, value-bounded text areas, ect...) and plotting widgets. While most GUI frameworks provide basic controls that can be customized into more complex fashions, thus covering the first use case, charts are not always supported, or lacks functionalities.

Evaluation Outline

Therefore, the evaluation has been carried on with special regard for this aspect.

CERN also have specific requirements for charting widgets. In some cases the amount of data to handle is very large, in some other cases the incoming rate of information is very fast. For this reason, their performance for refreshing speed and amount of point that could be handled at a sufficient frame rate (25 fps) was also assessed.

2. The *ease of development* of the interface (and the availability of RAD^[11] tools)

In our specific use case, the ease of development is a crucial aspect.

Our section is in charge of providing support, directives, best practices definition and operations expertise for a handful of supported frameworks, while the actual development effort is carried on by the Control Room's operators, that have the field expertise to make use of the data that comes from the backend. In the most common case they are, however, physicists and field experts, neither expert nor experienced in software development.

That said, the importance of providing them with easy to use tools should be much clearer. For example, we were especially careful in identifying possibilities for the developer to design the GUI entirely in a WYSIWYG^[116] fashion, by drag-and-dropping the controls on a canvas and binding them to the backend with ease. Such workflow would help them develop application in a quick and painless way, and at the same time to produce higher quality code which can be maintained, modified and extended by other people with ease.

The evaluation has to take into consideration all these aspects at the same time: the development process and the resulting application are both evaluated. This is aimed at capturing both the hard features of the framework, as well as the soft characteristics.

In order to be coherent, we defined some reference parameters to consider for each approach, the most important being (in relevance order):

1. The *architecture of the produced apps*: assess if the visual parts can be easily separated from the application logic, how accurately the MVC pat-

tern^[24] can be implemented, how intuitive and maintainable the resulting code is in the general case.

2. The *availability of all necessary widgets* that are required to visualize large and fast amounts of data, but also to control efficiently the machines.
3. The *performance of widgets*, especially of charting components. They are expected to display measurements at multi-Hz frequency and very large datasets (to several tens of thousands of points per update).
4. The *availability of GUI builder tools*, with special attention to whether they provide round-trip engineering features to support incremental development, and possibly drag-and-drop prototyping features.
5. The *availability of good online documentation* and community support.

2.3 The Benchmark Application

The selected strategy for the evaluation has been based on the development of a *benchmark application*. Such application has been developed to be as similar as possible on all the evaluated frameworks, in order to give a uniform perspective of the capabilities of each.

The benchmarks features a connection to the backend providing accelerator's data, in order to test it in a real life scenario, and the controls we want to evaluate, properly connected to the backend source in a way that loads them properly, especially in the case of plotting widgets.

The benchmark application is meant to have two good properties:

1. It should be straightforward to implement, for practical reasons
2. It should stress all the important sides of a real application development, in order not to overlook any important weakness of the evaluated approach or, even worse, to not spot potential show-stoppers in the evaluation stage.

Those two parameters defined the final design of the app.

2.3.1 The Interface

The interface of the test application, shown in the figures, was meant to test some of the most common used component in our control system's GUI.

First of all, basic widgets like labels were introduced, along with some basic customization such as color, font, font size, etc. Then, the interface was divided into two section by means of a tab pane: a crucial element in many applications, as the screen surface available for each app is often limited, and space usage must be maximized.

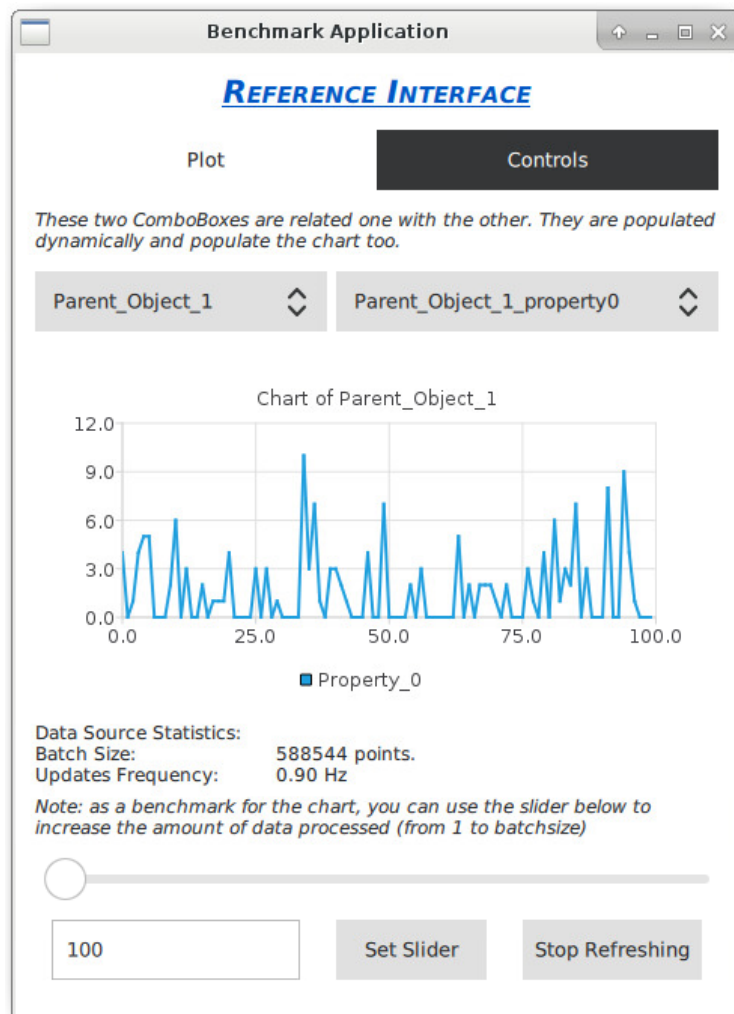


Figure 2.1: Tab "Plot" of the benchmark application

The tab pane has two tabs that focuses on different widgets. In the first one, the main component under inspection is the plotting widget, which is

2.3 The Benchmark Application

crucial with respect to the purpose of our evaluation. Alongside the plot, a slider was added to test basic plot interaction capabilities.

The slider is also crucial to benchmark visually the plot capabilities under stress. In fact, the data source plugged to the plot is one of the most demanding found in our backend, publishing batches of more than half a million points every second. Given that in some cases the plotting widgets were unable to deal with such a huge batch of data, the slider is introduced to throttle the incoming amount of points to a level that the widget can withstand.

In addition, the first tab features two combo boxes interacting with each other basing on the reciprocal selection, and some simple button to control the slider position and to toggle the data retrieval.

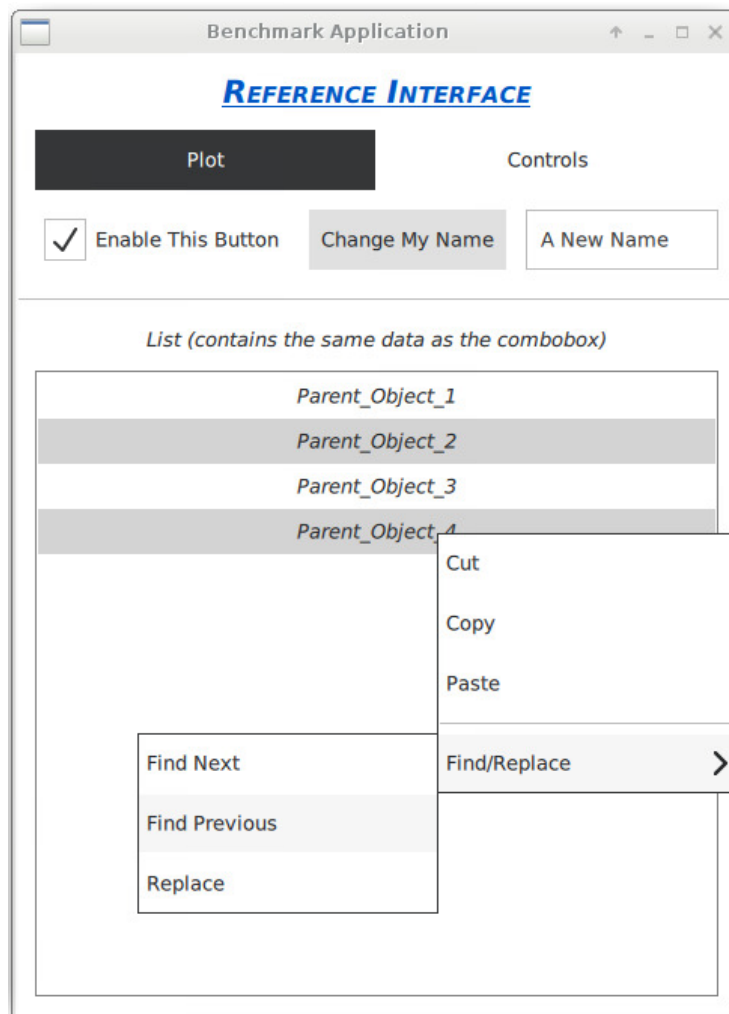


Figure 2.2: Tab "Controls" of the benchmark application

Evaluation Outline

In the second tab, basic interaction between widgets were tested in the upper part of the tab. To follow, a list widget is added, taking data from the same source as the first tab's combo boxes in order to test model decoupling from the widgets.

In addition, every list entry features a nested context menu, that triggers a small popup.

2.3.2 The Data Source

The data being rendered in the plot comes from actual equipment of the accelerator's infrastructure. This makes the benchmarks more meaningful with respect to our clients' use cases.

In addition, fetching data from real sources allows us to check whether the connection with the backend can be performed smoothly and, eventually, how to improve the integration with it. In some cases such integration was far from seamless, and it could be a reason for not adopting a specific framework, regardless of its scoring in the benchmarks.

The selected data source is an LHC device providing data and meta-data about the beam. The device publishes huge amount of data structured into properties and fields. Without diving into the details of the nature of the data itself, the specific array plotted in our benchmark is labeled `LHC.BSRTS.5R4.B1/Image#imageSet` and, despite the name, is a one-dimensional array of integer values. It contains a number of points that varies between 40 000 and 600 000 points, depending on the machine configuration, and gets updated at approximately 0.9 Hz.

Other devices publish smallest batches of data at much higher frequencies: for example, real-time data about the magnetic cycle of the injectors (SPS, PS, etc...) gets published at approximately 25 Hz. However, such sources publish one single point per update (the newly read value) and therefore were deemed less demanding in terms of performance.

2.3.3 Charting performance assessment strategy

Given the design of our application and the nature of the selected data source, we designed also a small performance assessment strategy for the plotting widgets.

2.3 The Benchmark Application

The test aims at assessing the maximum frame rate (frames per second, or FPS for short) that the application can possibly render while keeping the chart updated. The frame rate is deemed sufficient from a minimum of 25 FPS up, as it means that the interface is responsive and smooth, without showing any lagging behavior. Between 25 and 15 FPS the application lags, but is still responsive, while below 15 FPS it becomes unusable and, as we observed, in general crashes under the rendering load after a few minutes.

In order to assess the frame rate we made use of the slider component in the first tab. As said, the slider is used to throttle down the number of points to render and to allow us to manually find the maximum load the application can withstand. However, it also provides the possibility to test the widget more rigorously.

The test was carried on as follows. The application was started with one single series shown in the chart and the slider set to an initial, small value, and its maximum frame rate was recorded over an interval of 2 seconds. After that, the slider was moved forth of a fixed value and reassessed for the frame rate over two seconds. Once the slider reached its maximum value, or the application crashed, all the values were stored and another run was performed.

In practice, for each technology we performed this test 30 times and averaged the results. The initial value was set at 100 points and the increment was 100 points for less powerful charting widgets, and 500 for more powerful ones. During the week when the tests were performed, the data source was emitting an unusual small amount of points (approximately 50 000), which was still considered sufficient for an initial performance test.

The resulting data was plotted in a FPS/points chart showing average values and min-max boundaries, in order to assess also the variability of the results.

In addition, it is worth noticing that the tests were carried on on a virtual machine over a VNC connection. This was done for practical purposes: physical consoles connected to the accelerator's networks have a customized environment which made unpractical to run most of the benchmark applications, and not many of them were available for extensive testing at this point of the evaluation. VPCs instead could be managed easily, without involving too many external personnel, and could receive data just as the consoles would.

Some simpler applications were run on the physical consoles for different

tests and the performance difference was indeed not negligible. The results got from this evaluation are therefore to be considered slight underestimations of the actual performance of the software, but can still give very interesting and valuable insights of the scalability of their performance.

2.4 Selected Technologies

Each of the listed technologies has been evaluated separately and compared according to the same criteria.

They are:

- **JavaFX.** As said, JavaFX is the current framework my section provides support for. It is evaluated along with the others to provide a reference point for the final comparison.
- **Qt Jambi** The previous major version of Qt, Qt4, had a full fledged binding with Java called QtJambi. For this evaluation, we tried a few community ports of such binding on the latest major version of Qt.
- **Qt** Qt is a very mature and wide framework that support a large variety of GUI design styles. We evaluated the two main styles:
 - **QtWidgets** Qt allows developers to define GUIs through an XML file, that is parsed and converted into a source file that can be imported into the app normally. Application developed in this way are called Qt Widgets applications. It is the most common, stable and well-tested approach to Qt development.
 - **QtQuick (QML)** Qt also supports a declarative language, QML, for GUI design, and embeds a JavaScript engine that can be used to make the GUI reactive. Therefore, one can design a full fledged app without having more than a few lines of Qt to bootstrap the rendering engine, and developing the rest of the app in QML.
We tested QML GUI development in quite a peculiar way: instead of developing a proper C++ backend, we tried to embed also the logic in JavaScript, without relying on any "true" backend.

- **PyQt5** One of the most popular binding of Qt for Python is PyQt5. Given that our backend services partially exposes their APIs to Python, PyQt5 was considered worth an evaluation. Another point in favor of PyQt5 is the increasing interest in Python arising from our clients, as well as a general raise in the interest in this language in the scientific community, and the presence of a big community worldwide.

PyQt5 wraps Qt entirely. Therefore we could test it in two ways:

- **A QtQuick PyQt5 app**, trying to reuse the same QML file used for the original QtQuick benchmark app.
- **A QtWidgets PyQt5 app**, trying to reuse the same .ui file used for the original QtWidgets benchmark app.

2.4.1 Web Technologies

At this point the reader might have noticed one option that was left out: Web GUI technologies. "Going Web" is apparently the main trend for GUI in general, and the evaluation is also meant to provide more information for us to stand in favor or against the adoption of Web GUI technologies for the CERN control system's frontend.

Web GUIs are becoming more and more relevant as the paradigm moves from desktop-based application to cloud-based and mobile-based apps. Web GUIs are fast to develop, extremely flexible, run on every device supporting a browser and even, in some cases, without (for example, for ElectronJS^[105]). Finding experts for them is quite easy and, due to the fast learning curve, even most of our customer probably have experience with them, or in any case they would learn the technology quite fast.

In addition, being the current trend for interfaces, it is guaranteed that they will continue to develop new features and improve constantly in the next years, as well as their tooling and all the related technologies.

In our specific use case, Qt got priority for an in-depth evaluation for a number of reasons:

1. Web GUIs do not offer any serious advantage over any other non Java library: the binding issue still persists.

Evaluation Outline

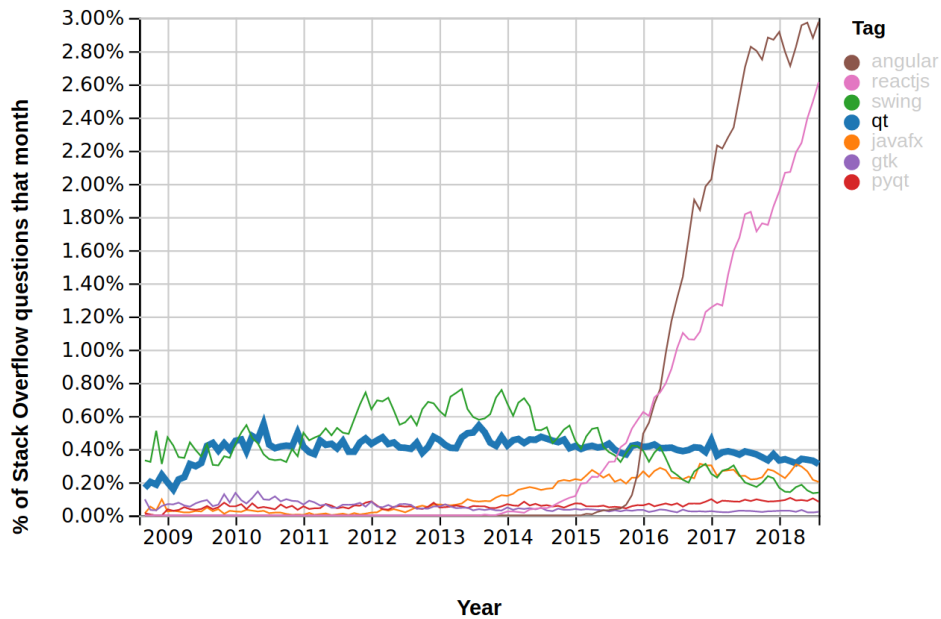


Figure 2.3: StackOverflow Trends^[4] over a 10 years period. Recent numbers for two popular Web GUI toolkit, AngularJS and ReactJS, completely annihilate desktop-based toolkits figures. Not only the exponential growth of the share of questions per month is staggering, but also the absolute amount of them.

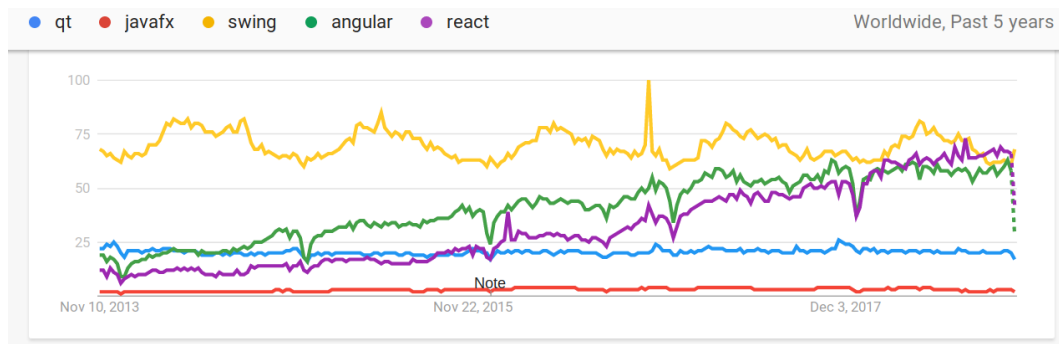


Figure 2.4: Google Trends^[5] over a 5 years period. While less impressive than in the StackOverflow Trends, both Web frameworks shows a solid and steep growth in interest that seems to have just passed even Swing's leadership. Is interesting to notice that this is a plot of Interest over Time, defined as "search interest relative to the highest point on the chart for the given region and time."

2. Web GUIs need a browser in order to run. This issue could be avoided by using the ElectronJS^[105] framework: a technology that allows creating desktop interfaces with JavaScript. ElectronJS is also renowned for being very hungry of memory, but given that no exploration was done on it, it cannot be told by experience.
3. Their main development language would be JavaScript, which is considered a sub-optimal solution. JavaScript, as the name implies, is a scripting language, interpreted and non typed, not meant to be used to build complex architecture.
4. The frameworks are short lived. Web technologies evolved very fast in these year, and even if they overall grow and improve, the single libraries often become obsolete after a few years only, getting quickly abandoned in favor of the latest ones. Of course such a quick pace is not suitable at all for control systems development. They must rely on very stable technologies that will be supported and running for years, potentially decades, with no need for replacement.
5. The tooling situation is extremely varied and short lived as well, adding the overhead of messy development environments, complex deployment chains, and so on.

The reader might argue that such issues are not any bigger than any of the drawbacks that adopting Qt may bring. Indeed an exploration of Web GUIs is not excluded *a priori*, but simply not deemed more urgent than any other. In addition, Qt looked much more promising at a first glance, giving us a strong push into further investigations.

Chapter 3

Current State: JavaFX

JavaFX was developed by Oracle as the ultimate Java desktop GUI framework. It has long been considered superior to its predecessors, like Swing and AWT, for its clear implementation of the MVC pattern^[24] and the effective decoupling of the visual description of the interface from the application logic.

3.1 Installation and setup

The installation process is quite straightforward. It goes in three main steps:

1. Install Eclipse (or any other Java IDE).
2. Install SceneBuilder^[28], the builder tool for FXML files (more on SceneBuilder in the Tooling section).
3. Install the e(fx)clipse plugin^[18] to wire the two together (or an equivalent plugin for the selected IDE).

With respect to the first point, CERN provides its users with a Secure Delivery Center that ships a customized Eclipse distribution and customized installer. One of the distributions also includes already the e(fx)clipse plugin, making the process even easier. In case of a custom install, the plugin can be installed easily through the Eclipse Marketplace.

SceneBuilder has still to be installed separately, but the process is almost trivial, as SceneBuilder is shipped as a standalone executable that does not even require an installation^[19].

3.2 Development process

JavaFX development does not add a lot of complexity over regular Java coding. The only relevant difference is the possibility to decouple the GUI design from its logic by making use of FXML files.

Best practices guidelines by Oracle^[27] suggest to implement an JavaFX application in this way:

- The View is entirely defined within the FXML file. FXML files are designed to be able to represent fully any JavaFX Scene Graph and all its possible features.
- The Controller is a specifically designed Java class whose name is set into the FXML file in a special attribute of the root node. It has access to the reference of all the graphic elements with the use of a special Java annotation, the `@FXML` tag. Its role is to translate GUI events and actions into the appropriate Java call.
- The Model is made of POJOs and is ideally decoupled by the View and the Controller, with the exception of property bindings.

An additional element, the Application, hosts the entry point of the JavaFX application, loads the FXML files, and accomplishes higher level operations like startup and shutdown management.

An example will clarify the point. Let's take our benchmark application's base layout: a label on top of the window, and a tabpane below it.

Such structure can be defined in two ways:

1. By instantiating explicitly in the Java code the JavaFX objects and defining their position into the Scene Graph:

```
1  BorderPane border = new BorderPane();
2  Label titleLabel = new Label("Reference Interface");
3  /* ... more label attributes set ... */
4  border.setTop(titleLabel);
5  Label centerPane = new TabPane();
6  /* ... create and add tabs ... */
7  border.setCenter(centerPane);
```

2. Or by describing them into an FXML file, with an XML-like syntax:

```
1 <BorderPane fx:controller="benchmark_app.views.
  RootLayoutController">
2   <top>
3     <Label text="Reference Interface" [... more
  attributes ...] ></Label>
4   </top>
5   <center>
6     <TabPane [... more attributes ...] >
7       <!-- content -->
8     </TabPane>
9   </center>
10 </BorderPane>
```

By designing the GUI through FXML, the framework pushes strongly the developer toward a pretty clean implementation of the MVC pattern. MVC (Model-View-Controller) is one of the most relevant architectural patterns for developing clear, reusable and maintainable application, and as such highly desirable, especially in the context of GUI development.

For a more complete example of a complete MVC implementation, please refer to the Annexes.

3.3 The application

The resulting application is pretty small and includes only 5 source files:

1. *MainApp.java*: the entry point of the application
2. *RootController.java*: the Controller of the application
3. *DataSource.java* and *ComplexObject.java*: the Model of the application
4. *RootController.fxml*: the View of the application.

The MVC implementation is here highlighted, but this is the default application structure suggested by most of the documentation and tutorials online, and also the most effective.

In this specific case, in order to keep the application simple, one single controller was implemented. However, the application could have been structured in an even less monolithic fashion by defining one root FXML and two children FXML, one for each tab. In this way, each tab would have had its own controller and could, in theory, be reused.

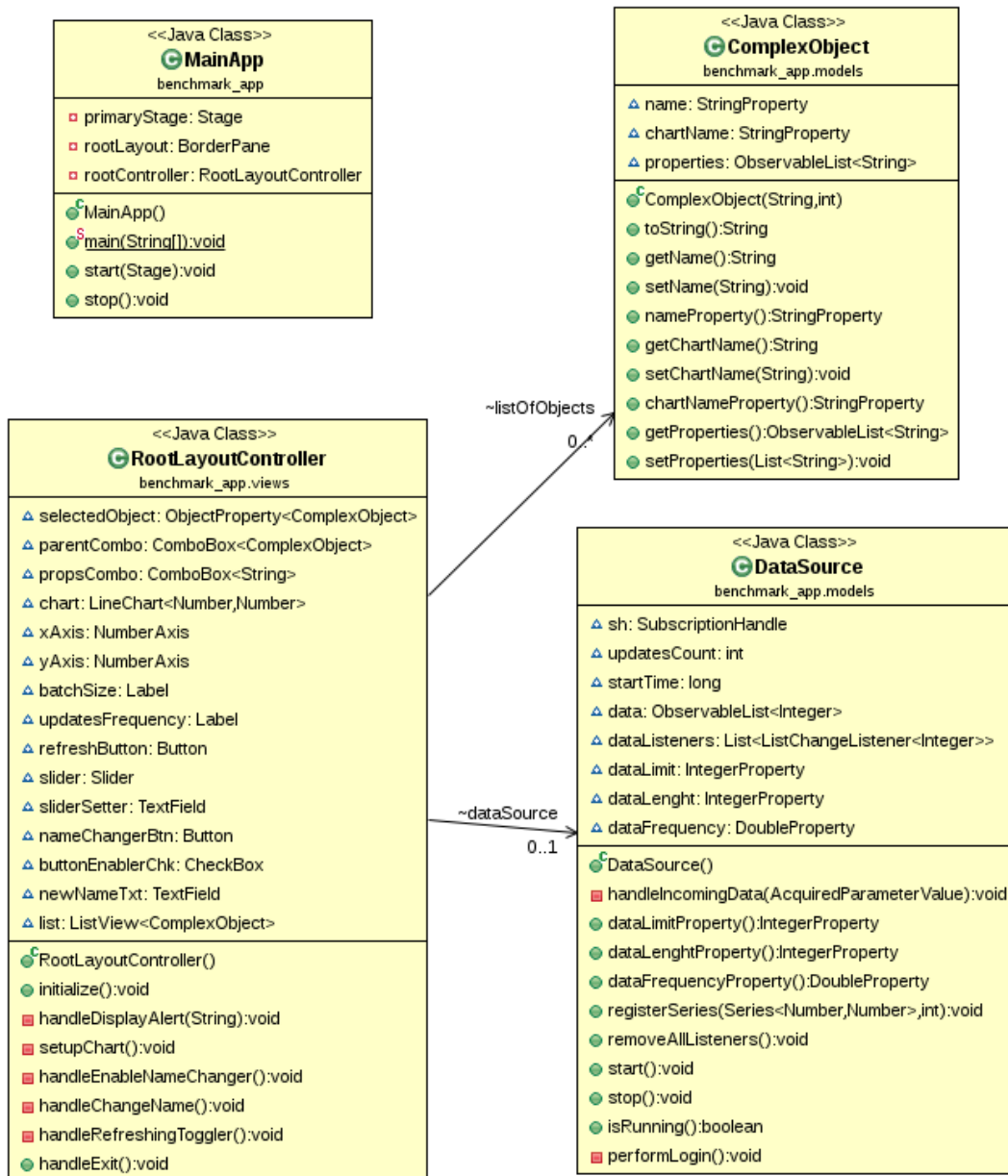


Figure 3.1: Benchmark App's Class Diagram for the JavaFX implementation.

3.3 The application

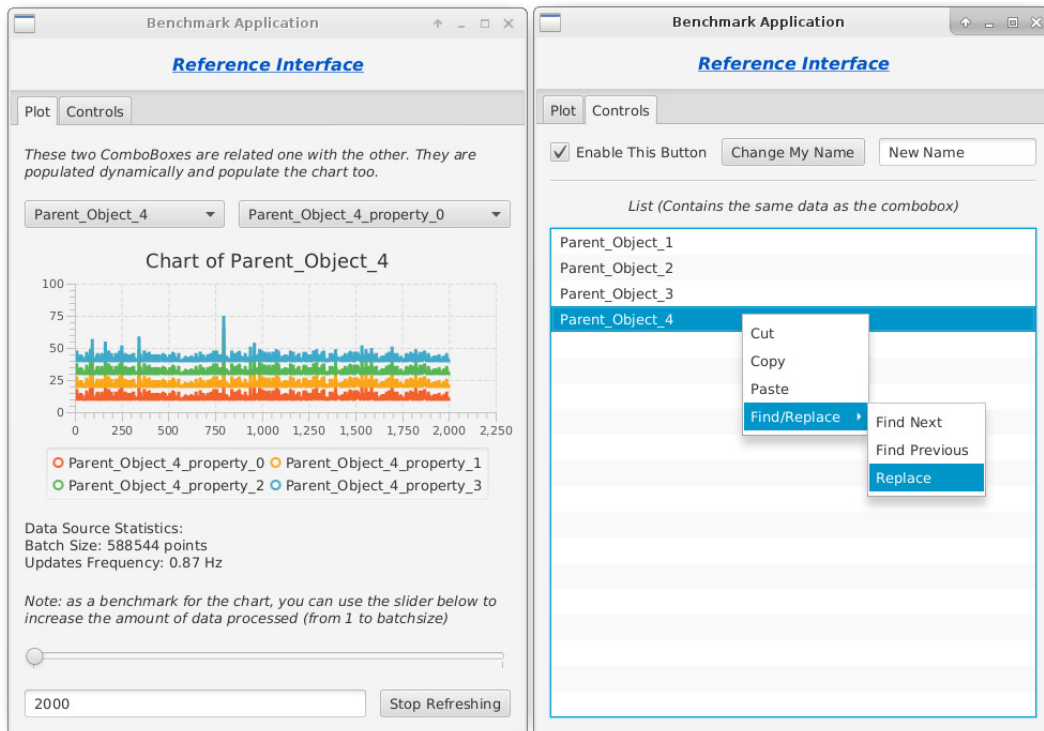


Figure 3.2: *Vanilla JavaFX applications have a pretty good looking interface, but they are quite poor with respect to charting.*

3.3.1 Charting

What is most surprising about the final product is the charting performance. A naive JavaFX application, developed according to the tutorials and howtos available online, ends up including a good looking, but low-performance chart. For four series, the chart could not be pushed into rendering more than 2000 points per series, which is almost 300 times less than the target data source’s size.

This problem is actually known. A paper^[29] from some former members of my section already deals with the matter of implementing all the features that are missing from JavaFX charts in order to meet CERN basic requirements.

With respect to the specific issue of the number of points, their solution is quite interesting: they simply reduce the number of points to be plotted using a smart algorithm that preserves the signal’s shape. This is done by extending the basic JavaFX `ObservableList` class into a `DataReducingObservableList` class that reduces the dataset every time it is updated. As for the paper, the reduction is performed using the Ramer-Douglas-Peucker algorithm, but

custom data reducing algorithms can be plugged in at need.

This is a good showcase of the amount of effort CERN already poured into JavaFX development. The extension library removes the problem of big data sources entirely and enables the users to visualize as many points as they receive.

It is worth noticing, however, that even extended, JavaFX charting still does not match what was possible with the previous framework, Swing. CERN has a very long history of GUI building in Swing. A very large number of custom components have been developed over the years to cope with different use cases, especially with regard to charting.

A very prominent example is the so-called JDataViewer library^[20]. It is a Swing library specifically designed to implement all CERN's required customization in terms of charting and, indeed, provides a staggering amount of features for very complex charting use cases^[21].

The current direction for GUI development, now under evaluation, was to proceed in this direction and make JavaFX closer and closer to its Swing implementation.

3.4 Tooling

In the case of JavaFX, the GUI design is made easier by a tool called SceneBuilder^[28].

SceneBuilder is a software that allows developers to design their GUI using a graphic interface, leaving to the software the task of generating an FXML file that can then be loaded into a JavaFX application. In a real case scenario, developers do not need to write any XML code, and may even not be really aware on how does it look like, because SceneBuilder takes care of its generation, validation, and binding with the relative Java backend.

From a user perspective, SceneBuilder qualifies as a RAD application, setting quite a high standard for competing frameworks under the tooling point of view.

In addition, our team developed a few custom integration to SceneBuilder, meant to provide an integration with our custom components.

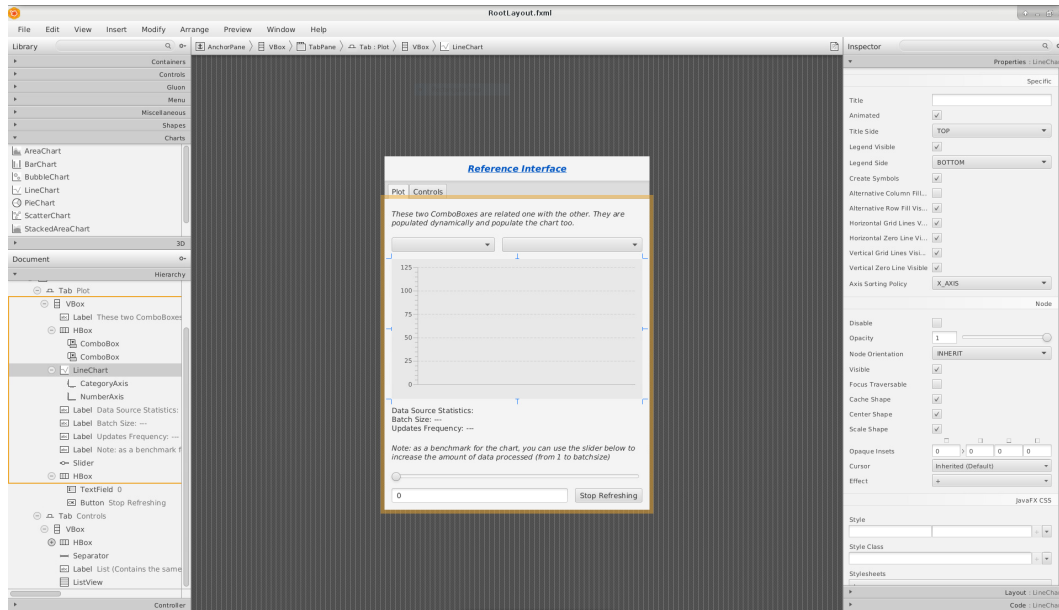


Figure 3.3: SceneBuilder’s interface.

3.5 Documentation

The documentation for JavaFX^[22] is in general complete and up-to-date. It is maintained by Oracle and integrated with a good number of tutorials and how-to-s, making the overall situation very positive. The Javadoc for method and classes is also quite complete and in general of good quality.

With respect to CERN’s JavaFX custom components, the situation is a little less positive. Javadoc is present, even if lacking for some components, and partial documentation and how-to-s are present in CERN’s Wiki pages, but it is still far from complete and, in some cases, even outdated or left undone.

3.6 Outcomes

JavaFX is strongly oriented toward a clean implementation of the MVC pattern, even if it does not enforce it. ”Old-style” GUI design, fully embedded in the application code, is still possible, even if discouraged.

This flexibility is very useful to help developers moving from a Swing-like approach to a more modern and decoupled architecture for GUI development. It also allows, at need, integration with Swing components, something that

Current State: JavaFX

proved almost necessary during CERN's migration to JavaFX, especially with respect to the JDataViewer library.

In general, JavaFX has no evident drawbacks with respect to CERN's requirements, apart from the foreseeable lack of support that is going to face. This makes its replacement very complicated in many ways: first of all, because of the difficulty to find another framework offering exactly the same features, and second, because it makes more difficult to justify with our clients the sudden decision to abandon a very good framework as JavaFX is.

Chapter 4

Qt: An Overview

Qt is a C++ cross-platform application framework and widget toolkit. It is used to create graphical user interfaces and applications that run on various software and hardware platforms, while still being a native application with native capabilities and speed. Being at the same time an application framework and a widget toolkit, Qt can be used to develop both native-looking GUI interfaces and command-line tools and consoles.^[30]

Qt supports various compilers, including the GCC C++ compiler and the Visual Studio suite and has extensive internationalization support. Other features include SQL database access, XML parsing, JSON parsing, thread management and network support.

Qt is currently being developed both by The Qt Company^[7], a publicly listed company, and the Qt Project under open-source governance, involving individual developers and firms working to advance Qt. Qt is available under both commercial licenses and open source GPL 2.0, GPL 3.0, and LGPL 3.0 licenses^[31]. In 2017, the Qt Company estimates a community of approximately 1 million developers worldwide in over 70 industries^[32].

Given those figures, it is easy to see how Qt can be considered a valid candidate as main GUI framework for CERN's interfaces development. In the next paragraphs I am going to outline the main characteristics of the framework and how it has been analyzed with respect to CERN's basic requirements.

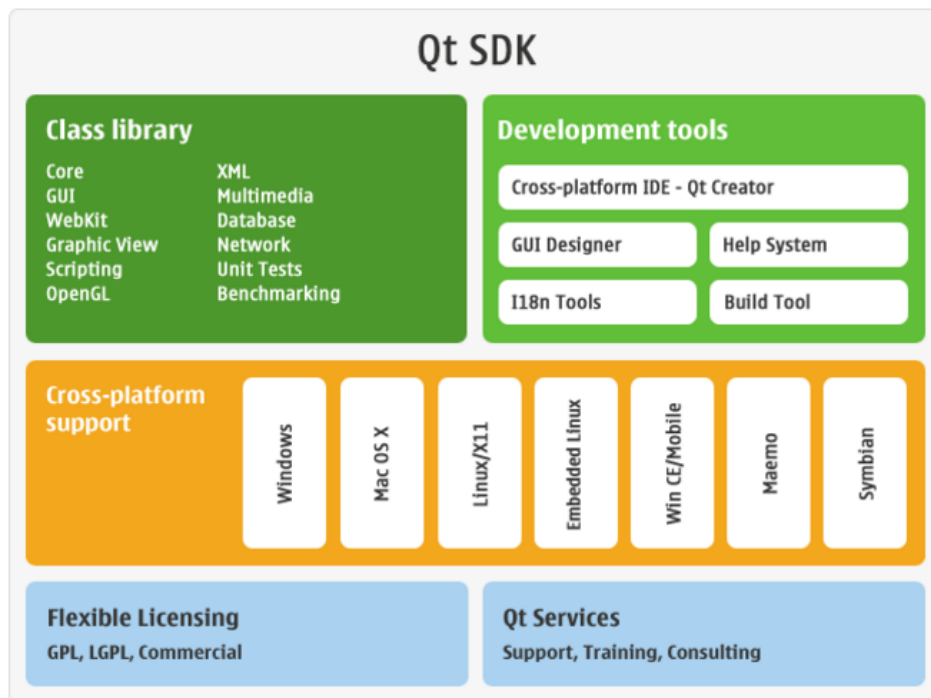


Figure 4.1: Qt5 SDK Overview

4.1 Framework architecture

Being a very large and comprehensive cross-platform framework, Qt comes with a peculiar architecture and some specific terminology which is necessary to understand, or even describe, some of its design choices. Here is a selection of those which were considered the most important for an introductory overview of Qt's structure.

4.1.1 QtQuick *versus* QtWidgets

Qt is a very mature framework but, in order to keep up with the latest trends, it has sometimes to perform some potentially heavy changes to its APIs, its design patterns, and its features. The latest major release indeed shows clearly the signs of a strong architectural shift with respect to its predecessor, while retaining the old APIs as a secondary, more old-fashioned way to develop Qt GUIs.

As of release 4, Qt focused on the approach of designing GUIs using XML files (with a `.ui` extension) for the layout and the static definition of the GUI,

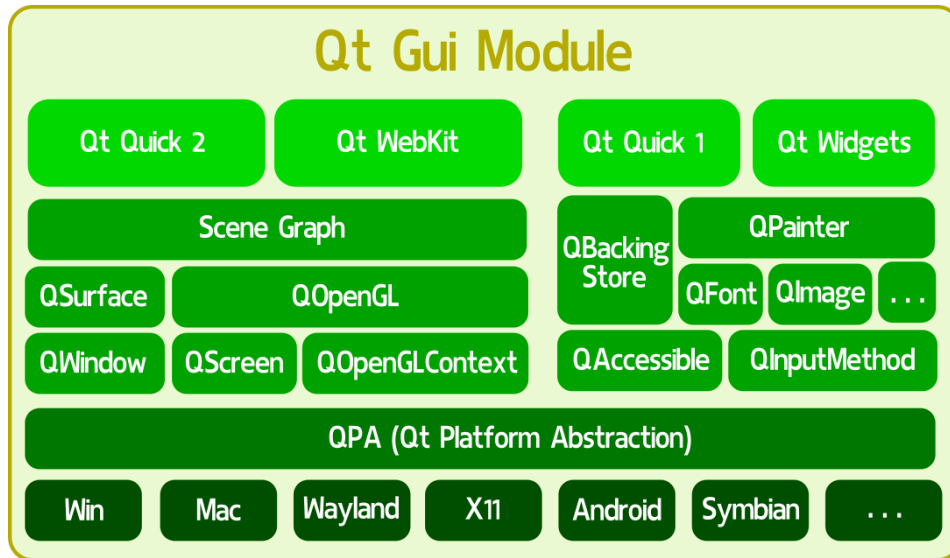


Figure 4.2: Qt Gui Module overview. There are two clearly separate stacks for GUI design: QtWidgets (with the old QtQuick1 implementation), and QtQuick2 (with the newly introduced QtWebKit), compatible with OpenGL.

just like JavaFX does. Such XML files are later compiled into source code that can be imported in the final application.

In Qt 5, instead, the XML solution was made secondary in favor of an even more decoupled solution (firstly introduced in Qt4.8), based on so-called QML files (extension `.qml`). QML files describe the interface in a JSON-like style, making the file very readable for developers and at the same time very easy to generate with a tool. These QML files also support scripting languages: therefore part of the view logic can be implemented directly in the QML file using Javascript code.

The final result is somehow similar to a Web interface: on the frontend side, a QML static file made reactive with JavaScript, and a C++ backend, almost completely decoupled, that communicates with the frontend both through explicit API calls and through a signal-slot mechanism typical of Qt^[38].

One can therefore make a clear distinction basing on the chosen GUI descriptor format. Applications that make use of `.ui` files are said to be *Qt Widget applications*. Instead, applications that make use of `.qml` files are said to be *Qt Quick applications*. Such a distinction will be very useful in the following chapters, as it is a recurrent choice developers have to make when

starting a new application.

4.1.2 Signals and slots

Signals and slots are possibly one of the most prominent features of Qt's architecture. They are language constructs introduced in Qt that makes easy to implement the observer pattern, which is heavily used to decouple effectively the interface design from the application logic.

The idea is that GUI widgets can send signals containing event information, which can be received by other controls using special functions known as slots. At the same time, the underlying model can send signals upon modifications, and such signals can trigger actions on the GUI, such as refreshes.

More on this pattern can be read in the documentation^[38].

4.1.3 The Meta Object System

One peculiar feature of Qt is the possibility of relying on a number of mechanisms that are generally unavailable to C++ developers. One of them is the so called Meta-Object System^[41]: a set of APIs that provide reflection-like capabilities in a way that reminds its Java implementation.

Reflection is "the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime"^[42]. In Java, reflection allows inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It also allows instantiation of new objects and invocation of methods by name (i.e. passing the name of the method, as a string, as parameter in a function call that will invoke that method).

In Qt, a similar behavior is obtained with the help of a separate tool, called the Meta Object Compiler (`moc`^[35]). The `moc` tool reads a C++ source file and it produces another C++ source file which contains the meta-object code for each of the classes found.

The resulting meta-object provides, first of all, the typical Qt's signal and slots mechanism described above. In addition to this, meta-objects provide features like introspection, asynchronous function calls, and many others.

The Meta Object System also allows QML interfaces to access C++ methods and properties directly. In fact C++ classes can be "registered" in the

Meta Object System^[33;34] and, if they comply some basic requirements (such as including the Q_OBJECT macro), they will be available to QML interfaces just as native QML components would.

4.2 Qt as CERN's GUI Framework

Now that the basic features of Qt have been outlined, let's examine closely the main issues we would face integrating it to CERN's software stack.

4.2.1 A C++ Framework

Qt is a C++ framework. This is, in itself, a major challenge to the integration between Qt and or backend: Java bindings to C++, and vice versa, are notoriously complex to setup, far from seamless, and prone to translation issues among the two languages. This leads to not-idiomatic APIs, complex error handling or lack of it, difficult or impossible memory management, carrying the constant risk of stumbling into dangling pointers generated by the Java garbage collector, and difficulties in passing objects between the two languages. Combining these issues with the inherent complexity of a software like Qt draws a concerning picture on the possibility of using a Qt GUI over a Java backend^[37].

4.2.2 Qt Bindings

Qt's powerful features has always attracted developers from other languages. Many companies were willing to create bindings regardless of the inherent complexity of the task, rather than developing new graphic libraries from scratch. In fact, Qt has a wide variety of bindings nowadays, the most popular ones being with Python and Go.

Regarding Java, a binding called QtJambi was being actively maintained as an open source project until 2015, when the release of Qt5 proved too labour-intensive for the community, that dropped the project. Nonetheless, the binding has been examined and evaluated as a viable option for the integration. Details of this evaluation are outlined in a separate chapter.

Python bindings instead showed good health, a lively community, and a lot of activity. The healthiest binding now available, PyQt5, is in fact about to

Qt: An Overview

be superseded by an new, official binding provided by the Qt Company (while PyQt5 is a community project), called *Qt for Python*, or simply PySide2^[49].

Being PySide2 released in technical preview in June 2018, we considered safer to test first PyQt5 and to keep an eye on the evolution of PySide2 and its adoption in the Python community.

In the following chapters I am going to detail how these parallel research direction progressed so far, and to which conclusions did each of them led.

Chapter 5

Qt over Java: QtJambi

GUI development in Java originally relied on Swing, but its limitations were evident. While Qt provided the possibility to model graphic interfaces with XML files, Swing has been lacking behind by providing only pure Java solutions, and therefore being less suitable for developing truly decoupled interfaces.

Because of this gap, a Java binding to Qt was considered very interesting by a number of companies, and so QtJambi was developed.

Originally backed by Trolltech, QtJambi was later adopted by Nokia and its development continued until 2009, when the project was open sourced. Qt was then at its 4.5 release. Despite the decline of desktop app's popularity, the community carried on the project until 2015, when Qt was already in its 5.6 version and QtJambi was still stuck to 4.8. Indeed, the port to a new major release proved too difficult with respect to the low interest in the project, that was abandoned.

Sporadic tentatives of resurrecting the project were made twice, first by Omix Visualization^[56], and later by Tili Labs^[57]. However, none of their codebases work out of the box on our local development machines, nor seem ready for production, so a more in depth analysis of the project and of these last attempts was necessary.

5.1 Installation and setup

Due to the lack of a working distribution, the installation and setup step was by far the most time-consuming process. It required us to investigate deeply

the internals of the binding, in order to fix it and port it to the latest Qt distribution.

The end user would not be required to go through these stages, once we manage to fix the binding code and provide a working distribution. In this case, we would design a specific installation procedure, depending on what is found necessary to make the binding work.

However, the process required a huge amount of time from our side and, in case we plan to provide our clients support for this binding, we would need to invest a sensitive amount of time in maintenance. In order to give a better idea of the complexity of the task, I will present a quick insight of the technology that runs the framework.

5.1.1 The QtJambi Generator

Java SDK provides a library called Java Native Interface (JNI) that allows Java developers to access native code running outside the Java Virtual Machine. Nonetheless, JNI is pretty complex to use and require a deep understanding on how both the native library (in this case, Qt) and the Java Virtual Machine work, in order to make them interoperate. Given the size of the Qt library and its complexity, a hand-made mapping showed simply impossible to consider, and also general purpose binders, like SWIG^[118], require a huge effort in order to produce any output.

Therefore the original developers of QtJambi created a tool, called QtJambi Generator^[59], to partially automate the process.

The QtJambi Generator is a tool that parses C++ header files to generate Java source files and a binding layer based on the Java Native Interface, that ties Java classes with their Qt/C++ counterparts. In order to generate such classes, it takes as input a typesystem specification and a header file. The header is used to generate a tentative Java conversion, following general rules. The typesystem instead is used as a "patch file" to correct both the generated Java source code and the underlying JNI calls.

While the generation is automatic, the typesystem file must be handwritten and should handle all the corner cases in which the generator may fail to produce meaningful output. This is indeed the most labor expensive part of the binding generation, as well as the most complex. An article from Qt

Quartely^[37] details all the underlying complexities of the binding, and how the typesystem is meant to cope with them.

Additional information and example on how the QtJambi Generator works can be found in the documentation^[62;63] on the matter, which also provide a few comprehensive examples of its usage^[64].

5.1.2 Actual installation process

The effort required to bring QtJambi to the modern era is in fact too big and requires a set of very specific skills that are missing in our team, namely: strong C++ coding skills (our team is composed of Java developers), deep understanding of the JNI and also a good understanding of how the generator itself works under the hood.

In order to compensate for this lack of know-how, I got in contact with the latest company who took care of modernizing the binding: Tili Labs^[60]. It is a small Canadian printing company who relies on Qt to provide their printing software interface while using a Java backend.

Their attempt to use QtJambi was indeed successful and provided a working binding between Qt5.6.2 and Java, over Windows 10 and Mac OS. Upon request, they also provided working binaries that could eventually be run by us as well. However, reproducing the same result on our Linux systems was not possible, neither it was to use the same procedure to provide a binding for more recent Qt releases (namely, 5.10).

Even the installation process of the provided Windows binaries was not straightforward.

The package came without the native libraries needed for it to run. In order to make it work, we needed to guess the necessary folder structure and the correct .dll to place nearby the provided jars. Such .dll came from a few different locations in our local Qt installation (which had to be the same exact version as the one they compiled QtJambi with) and were placed alongside the jars and into subfolders like "platform" and "plugin". We still are not completely sure we added all the required files, and whether some of them are redundant, as this implies a big number of trial-and-error runs of an application which is complex enough to trigger the entire library (and that we did not have).

For a more detailed description of the deploy layout, please refer to the Annexes.

5.2 Development process

Once the underlying software has been setup, QtJambi coding does not add too much complexity over regular Java coding. Just like in JavaFX, QtJambi offers different options to the developer: a monolithic Java-only definition of the GUI, or its decoupling through an XML file, or even the possibility to use QML.

This is, at least, what the documentation says. Our binding was not actually able to load QML files, because the providers of the binaries did not need the QtQuick module and did not port it. We could not get to have a working version of the `juic` compiler to try generating Java code from Qt Designer. Eventually we could get only a small, Java-only minimal demo of a working application, as a proof that the Qt core internals were actually ported: but nothing more.

If the binding was being correctly generated, the overall experience would have been quite fluid: however, such a result could not be achieved.

5.3 The application

The only application we could produce was nothing more than a minimal "Hello World" example application. Developing anything close to our target benchmark was eventually impossible.

The resulting application is limited to the following small snippet.

```
1 package qtjambi5;
2
3 import org.qtjambi.qt.gui.QFont;
4 import org.qtjambi.qt.widgets.QApplication;
5 import org.qtjambi.qt.widgets.QPushButton;
6 import org.qtjambi.qt.widgets.QVBoxLayout;
7 import org.qtjambi.qt.widgets.QWidget;
8
9 public class QtJambiHelloWorld extends QWidget {
10
11     public QtJambiHelloWorld(){
12
```

5.3 The application

```
13 // Setup the button "Say Hello"
14 QPushButton hello = new QPushButton("Say Hello!", this);
15 hello.setFont(new QFont("Times", 18, QFont.Weight.Bold.
16 value()));
17 hello.clicked.connect(this, "sayHello()");
18
19 // Setup the button "Quit"
20 QPushButton quit = new QPushButton("Quit", this);
21 quit.setFont(new QFont("Times", 18,
22 QFont.Weight.Bold.value()));
23 quit.clicked.connect(QApplication.instance(), "quit()");
24
25 // Setup the window
26 QVBoxLayout layout = new QVBoxLayout();
27 layout.addWidget(hello);
28 layout.addWidget(quit);
29 setLayout(layout);
30 setWindowTitle(tr("Hello World!"));
31 }
32
33 // Callback to react to user inputs.
34 public void sayHello() {
35     System.out.println("Hello!");
36 }
37
38 // Starts the application
39 public static void main(String args[]){
40     QApplication.initialize(args);
41     QtJambiHelloWorld widget = new QtJambiHelloWorld();
42     widget.show();
43     QApplication.execStatic();
44 }
```

Listing 5.1: HelloWorld.java

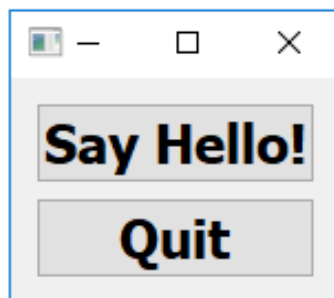


Figure 5.1: Nothing more than a very small Hello World example was considered reasonably possible with this library.

5.4 Tooling

In theory, the tooling for QtJambi is the same for standard Qt development (which will be described in more detail in the following chapters). QML files, if they could be used, would have been developed with QtCreator; .ui files could be edited with Qt Designer; while the actual coding could be carried on in any Java IDE, for example Eclipse.

The integration among the tools however is almost non-existent. A plugin for Eclipse was available^[65] but is now completely unusable.

It is known that the `uic` compiler was ported into its Java counterpart, the `juic` compiler, that however we could not get to work with Qt5.

A deeper inspection of the tooling was not considered worth, due to the state of the software itself.

5.5 Documentation

What is truly impressive about QtJambi is the accuracy of the documentation. Being a dead project, one could expect the documentation to be the first thing to die out: instead, one can still find plenty of tutorials, how-to-s, examples, and a couple of official websites still available: the old official documentation^[55] and the community one^[54].

The amount of available documentation was actually one of the main reason why the inspection of QtJambi was carried on in the first place.

5.6 Outcomes

The preliminary inspection of the state of the software confirmed the impression of a well developed, solid binding among Qt4 and Java. The documentation is still present and accurate, and a number of tutorials exists and are up-to-date with the latest release (with a few discrepancies). The tooling also was solid: a specific plugin for the XML designer was developed in order to compile the .ui files with the `juic` compiler^[58] (the Java counterpart of the `uic` compiler^[40]), building a comfortable development environment which matched closely the C++ one.

Qt Home · Examples

Qt Jambi Reference Documentation

Qt is the de facto standard C++ framework for high performance cross-platform software development. Qt Jambi is the Qt library made available to Java. It is an officially supported technology aimed at all desktop programmers who want to write rich GUI clients using the Java language, while at the same time taking advantage of Qt's power and efficiency.

The technology provides new possibilities for both Java and C++ programmers: It enables Java developers to take the advantage of Qt's features from within Java Standard Edition 5.0 and Java Enterprise Edition 5.0 as well as later versions. In addition, Qt Jambi also enables C++ programmers to easily integrate their Qt code with Java by providing the Qt Jambi generator.

Getting Started	General	Developer Resources
<ul style="list-style-type: none"> Installation Eclipse Integration Examples 	<ul style="list-style-type: none"> About Qt About Us Frequently Asked Questions 	<ul style="list-style-type: none"> Qt Jambi and Qt Mailing Lists Qt Community Web Sites Qt Quarterly How to Report a Bug Other Online Resources
API Reference	Core Features	Key Technologies
<ul style="list-style-type: none"> Qt Jambi API Javadoc Type System Qt Jambi System Properties Deploying Qt Jambi Applications Technology overviews Qt C++ Reference Documentation Qt Widget Gallery 	<ul style="list-style-type: none"> Layout Management Paint System Resource System Signals and Slots Thread Support in Qt Jambi Internationalization 	<ul style="list-style-type: none"> Style Sheets
Add-ons & Services	Tools	Licenses & Credits
<ul style="list-style-type: none"> Qt Solutions Partner Add-ons Contributed Qt Components 	<ul style="list-style-type: none"> Qt Designer Qt Linguist Generator JUIC 	<ul style="list-style-type: none"> Third-Party Licenses Used in Qt Other Licenses Used in Qt Support Training


Any problems encountered with Qt Jambi should be reported using a bug report. Include as much information as you can about your environment and versions. If you want to join the Qt Jambi users' mailing list, send a mail to qtjambi-interest-request@trolltech.com containing the single word "subscribe". Other inquiries can be directed through the regular channels.

Copyright © 2009 Nokia Corporation and/or its subsidiary(-ies) Trademarks Qt Jambi 4.5.2_01

Figure 5.2: The official documentation

Home Planet Users Contribute Community Documentation Downloads

Qt Jambi

The Qt library for Java 

Qt Jambi

Qt is the de facto standard C++ framework for high performance cross-platform software development. Qt Jambi is the Qt library made available to Java. It is an open source technology aimed at all desktop programmers wanting to write rich GUI clients using the Java language, while at the same time taking advantage of Qt's power and efficiency.

The technology provides new possibilities for both Java and C++ programmers: It enables Java developers to take advantage of Qt's features from within Java Standard Edition 5.0 and Java Enterprise Edition 5.0 as well as later versions. In addition, Qt Jambi also enables C++ programmers to easily integrate their Qt code with Java by providing the Qt Jambi generator.

For more comprehensive description of what qt-jambi provides, see [here](#).

This is new website released at 10.03.2012 after far too many delays. If you still want to see old website, it can be seen at <http://old.qt-jambi.org>.

Downloads

Windows	Linux	Mac OS X
32 bit: 4.7.1-beta3	32 bit: 4.7.0-beta2 64 bit: 4.7.0-beta2	32 bit intel: 4.7.0-beta1

Blog

Figure 5.3: The community documentation

Qt over Java: QtJambi

However, although the binding itself was well developed, it does not work at all with the latest version of Qt. In addition, there is no community behind it at the moment, which makes its adoption a risky choice for a company that intends to rely on this technology for many years to come. Therefore, even if the inspection of QtJambi and its compatibility with different Qt5 releases started with good expectation, it failed to produce any result, and has to be considered negative.

Chapter 6

Qt Widgets

The most basic way to design interfaces in Qt is by defining them in the code. Such an approach is very solid and gives the developer full control over the application, but it gives also a lot of responsibility into building well-structured apps and is quite a time consuming process, because in order to have a preview of the interface, the entire application has to start up.

In order to offer a practical solution to prototype the interface, Qt offers a way to design GUIs in a graphical way, without writing any code. This technique did not mean to be very far from the original, code-heavy way to generate interfaces: it simply generates the necessary code from an XML description of the interface, which was in turn generated by a tool.

Applications produced with this XML-based approach are called Qt Widget applications and are developed entirely in C++. Qt Widget applications are probably the most stable way to develop Qt-based interfaces, and the one that offer the highest guarantees for the future.

6.1 Installation and setup

The installation process of Qt is in general straightforward. Qt provides many distributions to suit different needs: online installers, offline installers, source packages, and a few other distributions for more specific use cases^[43;44]. Pre-built binaries comes for all the supported platforms, Android included, while the source install is modular and allows users to customize heavily the Qt installation to their needs.

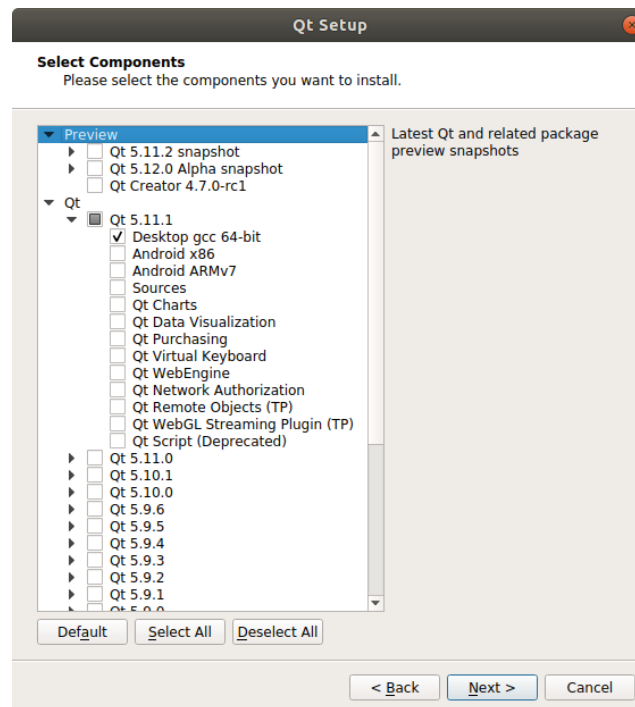


Figure 6.1: Qt’s installer can also be used after the installation to add, remove or update Qt components, given that a network connection is available.

For this application we used one of the mainstream way of installing Qt, that is the offline installation. The online installation was not possible in this specific context, as the machines that can access accelerator’s data are often not connected to the Internet, but only to the so-called CERN’s Trusted Network.

Once the offline installer was moved onto the target machine, the installation process consisted simply in following the wizard’s instructions. It also allowed some customization of the distribution, even though to a much more limited degree than a source install, and forced the installation of QtCreator, which is not strictly needed for Qt Widget applications.

In the future we plan to provide Qt and all the required tooling pre-installed and pre-configured on every development machine, in order to have a standardized environment to offer support for.

6.2 Development process

The development process, as expected, was much more painful than with any of the other approaches under test. Being a team of Java developers, none of us was especially skilled in C++ development, especially in using the latest C++11 and C++14 features Qt offers support for.

In addition to this difficulties, Qt is a very complex and large framework that goes as far as defining custom "visibilities" for class methods, that can be defined `private`, `public`, but also `signals` and `slots`. The number of macros involved in standard Qt development is also very high and, having no expertise in the domain, understanding them was far from trivial and sometimes lead to some kind of "cargo-cult" behavior from our side, for example for the "all mighty" `Q_OBJECT` macro.

The following header, coming from the `DataSource` class, should give an idea of which kind of challenges Qt development presents to its users.

```

1  #ifndef DATASOURCE_H
2  #define DATASOURCE_H
3  #include <time.h>
4  #include <QtCore/QObject>
5
6  class DataSource: public QObject {
7      Q_OBJECT
8      Q_PROPERTY(int* data MEMBER m_data NOTIFY dataChanged)
9      Q_PROPERTY(float updates_frequency READ updatesFrequency)
10     Q_PROPERTY(bool update_series MEMBER m_update_series NOTIFY
11         updateSeriesChanged)
12
13     private:
14         int* m_data;
15         bool m_update_series;
16         int m_datasource_updates_count;
17         time_t m_start_time;
18         float updatesFrequency() const;
19
20     public:
21         explicit DataSource(QObject *_parent = nullptr);
22         int* data();
23         void dataReceived(int* newData);
24
25     signals:
26         void dataChanged(int* newData);
27         void updateSeriesChanged(bool newUpdateSeries);
28 };
29 #endif // DATASOURCE_H

```

Listing 6.1: DataSource.h

6.2.1 Qt's Meta Object System

One of the first interesting features of the above code is probably the `Q_PROPERTY` macro. It is used to add custom Qt properties to the `QObject`: that is, members that emits signals upon changes. A preliminary description of Qt properties, signals and slots concepts was given in the Qt introduction in Chapter 4, and these properties will also be used heavily in the QtQuick application, but here we can see the very definition of it in the code.

The syntax of the `Q_PROPERTY` macro can vary a lot. In the general case, a property gets defined by giving it a type and a name. Usually it is also associated with a member of the class with the `MEMBER` label, but this is not always the case: for example, the `updates_frequency` property has no member, but it gets recalculated when required. To define this, it gets a `READ` label and passes to the macro a pointer to the "getter" function. Editable properties can provide also a setter function with the `WRITE` label. Many other labels can be passed to the macro: for more information, please refer to Qt's documentation on the matter^[39]

Interestingly, the last `Q_PROPERTY` macro has a `NOTIFY` label that points to the `DataSource::updateSeriesChanged()` method which, in turn, lays under the `signals` keyword in the header's body. This brings us to the next interesting point: Qt's custom keywords.

Signals are specific features of the library that relates to Qt's most powerful feature, the Meta Object System. It was also already outlined in the introduction to Qt and indeed is an extremely complex system to understand for a non C++ developer. To understand it, one has to go beyond simple code reading, and has to pay attention to the build process itself.

The build process of a Qt application starts earlier than in a normal C++ application. The first process to run is `qmake`, a tool that comes with the Qt installation and automatically compiles a Makefile for the app. After `qmake` has run, `make` processes the resulting Makefile which, in turn, does a lot more than simply compiling and linking the application. It sends all the header files through the Meta Object Compiler (`moc` for short), which generate some more source code files, the so-called *meta objects*. These meta objects are produced by processing all Qt's custom keywords and macros, and their purpose is to handle all the wiring between signals, slots, and properties. Only when all

these steps are complete, all the source code is compiled, linked and eventually executed.

While in QtQuick this machinery is mostly hidden (as we will see in the dedicated chapter), in Qt Widgets applications it is much more directly exposed. This puts developer in the position of having to understand how the meta object's code gets generated and, when it fails to compile, how to modify their own source code in order for the Meta Object Compiler to produce valid C++ code.

6.2.2 GUI Design

What we described so far is just the building process of a model class.

As said in the introduction, the actual GUI is not designed by writing code, but with a graphical tool called Qt Designer, which we will describe better in the dedicated section. Qt Designer is very straightforward to use to design the graphical side of the interface: by drag and dropping widgets on a canvas, the developer can easily generate an XML file that is compiled into C++ code by a tool called `uic` (User Interface Compiler) just before `qmake` is run.

However, Qt Designer cannot fully design the behavior of the interface, that must be coded by the user. In order to bind the interface to the backend, the developer has to extend the `MainWindow` class generated by `uic` and write code that, also in this case, to leverage heavily another Qt's core feature: the `connect` method of `QObject`s.

```
1 // Name-changing button
2 connect (
3     ui->pushButton_2, &QPushButton::clicked,
4     [&]() {
5         ui->pushButton_2->setText(ui->lineEdit->text());
6     }
7 );
```

Listing 6.2: Snippet from mainwindow.cpp

The Qt documentation tends to describe `connect` statement as a quite simple feature. Their aim is to connect an "emitter" object to a "receiver" object by connecting a signal to a slot. Therefore, the usual syntax of a `connect` statement should be:

```
MainQObject::connect( QObjectA, QObjectA::aSignal, QObjectB,  
QObjectB::aSlot );
```

However, often developers want to connect an object to a method that is not a slot, or that is not compatible (for example, it takes a `QString` while the signal emits an `int`). In this case, as shown above, the syntax becomes:

```
MainQObject::connect( QObjectA, QObjectA::aSignal, []() { /*  
... code of the lambda ... */ } );
```

However, lambda coding in C++ is far from trivial for inexperienced developer and, in our case, made the entire wiring process extremely time consuming and nearly impossible to debug. We had, eventually, to ask support to C++ developers from other departments, to give at least some general advice on how to properly use lambda expressions and function pointers in a way that was, at the same time, compiling in our application and in the moc-generated source files.

6.2.3 Build Process

To sum up all these steps, every time users want to build the application they have to:

1. Run `uic` to compile the `.ui` files generated by Qt Designer into C++ code.
2. Run `qmake` to parse the project folder and generate a proper Makefile.
3. Run `make` to launch the build process, that in turn:
 - (a) Invokes the `moc` to generate the meta objects for all the header files found
 - (b) Compile and link all the generated sources
4. Run the application

None of the above step is trivial and each of them can fail in obscure ways, making the development experience time consuming and, for a beginner, especially frustrating. These aspects may sound completely secondary, but the reader should consider that we are supposed to ask non-developers to produce their GUIs using this technology, and in the general case, they have pretty weak coding skills and no C++ experience at all.

6.3 The application

Given the problematic development process and the lack of expertise in C++ development, it should be no surprise for the reader to learn that we failed to produce a fully working application in the allocated time. A lot was learned from the process and most of the progress was actually made in the very last days of development. However, the learning curve was absolutely too low and, even at that point, the development was too slow and error-prone to be considered viable for non-programmers to learn.

This also prevented us from properly assessing the performance of Qt's charting library with Qt Widgets, which was the one supposed to be the fastest. However, as we are going to describe in the chapter dedicated to PyQt5, we managed to obtain a working Qt Widgets application through the Python binding and assessing that one for its charting performance. Assuming its C++ version can reach the same performance, if not even higher, charting in Qt Widget application poses no performance problems at all.

6.4 Tooling

As said in the Installation and Setup section, Qt enforces the installation of its own IDE, QtCreator. However, the only tools which are really required for Qt Widgets development are *Qt Designer* and the *UI Compiler*

Qt Designer is a software whose sole purpose is to allow developers to design their interfaces by drag-and-dropping widgets on a canvas. It generates automatically an XML definition, the `.ui` file. This file can be then processed by `uic`, the User Interface Compiler, to generate a source file that can be included in the application and extended at will.

From this perspective, Qt Designer offers a much more limited set of features comparing to QtCreator: more than an IDE, Qt Designer is more like a simple editor for `.ui` files. However, this approach provides more flexibility: indeed Qt Designer can be easily paired with any modern C++ IDE to act as an external editor, leaving the developer free to use their own favorite C++ development environment to create Qt application. From this perspective, Qt Designer resemble closely JavaFX's SceneBuilder.

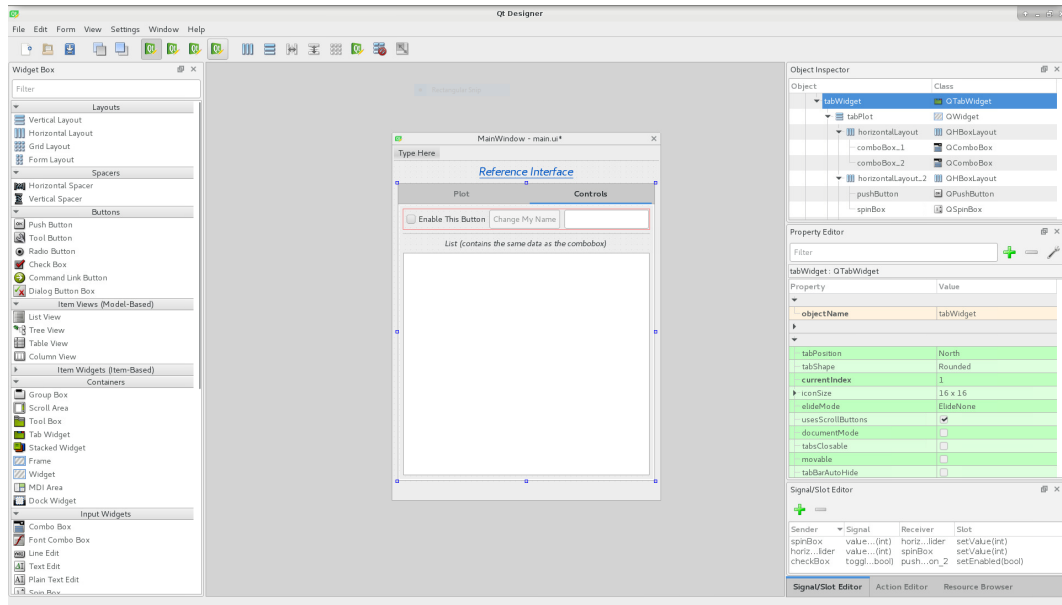


Figure 6.2: QtDesigner's interface

6.5 Documentation

In general, Qt documentation is very complete, comprehensive and up-to-date. Even considering the size of the API and the amount of features to be documented, the documentation is easy to browse and well supported by the community itself, that adds a huge amount of unofficial tutorials and Stack-Overflow answers to the already exceptional docs.

Content-wise, nothing crucial was found missing: the doc includes tutorials for all the tools and integration with many third-party software, best practices, and features a very large quantity of examples and commented examples.

A Qt Wiki^[47] also exists, but is in general a little more outdated and messy.

6.6 Outcomes

Qt Widget's evaluation produced somehow contradictory results. The technology itself is very solid, powerful and stable. Documentation is comprehensive, complete and accurate. The tooling is complex, but works smoothly and is open to customization and integration with external tooling, providing a few different, equally working workflows. The installation process is straightforward and every step of the development, installation, maintenance, deploy-

ment, are covered both by the documentation and by extensive amount of forum threads and StackOverflow questions.

However, the problem is on our side: we have no C++ development skills and it is not reasonable to assume that our users would acquire such skills either. While the hard features of the framework are exceptional, the Qt Widgets approach lacks almost all the soft features required for a successful integration into our software stack: not only from a technical perspective, but mainly for a "human" factor.

For this reason, we concluded that a pure C++ Qt development of our GUIs is not a valid replacement for JavaFX-based GUIs development.

Chapter 7

Qt Quick: QML & JavaScript

Apart from the classic Qt Widgets approach, Qt offers more decoupled ways to implement GUI applications.

The QtQuick technology aim to achieve perfect decoupling between the interface and its underlying model. In order to reach this goal, they are made of some C++ classes, building up the model, and QML declarative files that describe the GUI. At startup, these QML files are processed by a C++ component called QML engine^[81] that translates them dynamically into a user interface. In this way the developer obtains an architecture which adheres pretty well to the MVC pattern:

1. a declarative QML View, with its own internal wiring written in JavaScript,
2. a C++ Controller exposing slots that can be called from the view,
3. a reactive Model that exposes properties and emits signals upon changes.

This is indeed the suggested way to use the framework and the one which offers the most options to developers. However, another design can be envisioned on top of these three main elements: one which relies much more heavily upon the QML engine and its JavaScript execution capabilities.

The original purpose of JavaScript support in QML is to implement internal reactive behaviors in the interface without cluttering the backend with any pure frontend logic, like animations and graphic feedback. However, JavaScript can be leveraged to execute much more complex code, as the engine can execute any arbitrarily long script. It even supports external libraries imports^[82].

Therefore, we considered worthwhile exploring the possibility of embedding the entire application logic into a well-structured architecture of QML and JavaScript code, and then to eventually provide an interface between JavaScript and Java.

Such a design would provide us with a few advantages, especially from the perspective of the ease of development, because the users are no more required to code the backend of their applications in C++, but in QML and JavaScript. Concerns instead focuses on the actual performance that the QML engine could achieve under a load it was probably not designed for.

7.1 Installation and setup

The installation process is exactly the same seen for Qt Widgets applications: the same distribution can be used to develop both kind of applications. The only difference is that QtQuick requires some extra modules and components, but in a default installation process (that is, using pre-build binaries for the target system) the degree of configuration of the installation does not allow to add or skip such low-level components.

For this application we used the same installation used to develop the Qt Widgets application: in fact, having installed Qt with the offline installer, we could not prevent it from installing also all the Qt Quick tools and components.

7.2 Development process

The strongest advantage of QtQuick development lies in the development process itself. Being a declarative, JSON-like language, QML is very readable, easy to hand-write and to generate alike, and has a pretty comfortable learning curve comparing to many other programming languages.

A very small example will showcase the point. The following snippet simply defines the structure of the main window of the benchmark application, that is, the window that contains the top label and the tabs. Some property settings were omitted for brevity.

```
1 import QtQuick 2.9
2 import QtQuick.Window 2.2
```

```
3 import QtQuick.Controls 2.4
4 import QtQuick.Layouts 1.11
5
6 Window {
7     id: window
8     visible: true
9     width: 480
10    height: 640
11    title: qsTr("Benchmark Application")
12
13    Label {
14        id: labelTitle
15        color: "#0057c4"
16        text: qsTr("Reference Interface")
17        /* ... other properties ... */
18        anchors.left: parent.left
19        anchors.leftMargin: 13
20        anchors.top: parent.top
21        anchors.topMargin: 13
22        anchors.right: parent.right
23        anchors.rightMargin: 13
24    }
25    TabBar {
26        id: tabBar
27        /* ... anchors ... */
28        TabButton {
29            id: tabBtnPlot
30            text: qsTr("Plot")
31        }
32        TabButton {
33            id: tabBtnControls
34            text: qsTr("Controls")
35        }
36    }
37    StackLayout {
38        /* ... anchors ... */
39        PagePlot {}
40        PageControls {}
41    }
42 }
```

Listing 7.1: Main Window QML Definition

The snippet shows a few interesting features of QML. The most evident one is its readability, especially with comparison to an XML file. With the readability comes clearly also the ease of coding it by hand: editing QML files requires almost no code completion at all.

7.2.1 Layout System

Other more technical features are showcased here. One is the layout system, which is based on the concept of *anchors*. The anchoring system is just one

of the many layout techniques available, but is the one that provides at the same time more control and flexibility. It assigns to every widget 6 edge-shaped anchors (top, left, bottom, right, vertical center, horizontal center): each of them can be paired with any other anchor in the scene by fixing their distance. The achieved result is that, on window resize, widgets grow or shrink in a meaningful way, while their relative distance stays fixed.

Such a layout style is very effective: however, we noticed that it is not as straightforward to digest for developers that come from XML based layout systems. JavaFX and, as we will see, QtWidgets, are based on the concept of layouts as containers that automatically rearrange and resize their children at need. Such containers are nested in order to achieve a grid layout, or even more complex configurations.

In QML, this overhead would reduce strongly code readability, and so it was replaced with the anchoring system. This paradigm shift is indeed one of the only difficulties that former JavaFX developers might face while learning QML.

7.2.2 QML Components

In the snippet is also clearly visible how QML manages its own components and custom ones. The import statements for example are heavily modularized in order to let the users import only what is strictly needed in their script.

However, the most interesting point regards custom components loading. At line 30-31 QML declares two custom components that represent the tab's pages, `PagePlot` and `PageControls`. They are not imported anywhere and do not need so, because the QML Engine automatically imports all QML files that are in the same folder as the executed ones, and let the user load them by simply stating their name.

Other import strategies are of course available, but this is a clear example of how minimal QML code tends to be.

7.2.3 JavaScript Code

In the above snippet there is not any JavaScript. The usual way of embedding JavaScript looks like the following:


```

1 Button {
2   id: buttonRefresh
3   text: qsTr("Stop Refreshing")
4   onClicked: {
5     if(buttonRefresh.text == "Stop Refreshing"){
6       buttonRefresh.text = "Start Refreshing";
7     } else {
8       buttonRefresh.text == "Stop Refreshing";
9     }
10    dataSource.update_series = !dataSource.update_series
11  }
12 }

```

Listing 7.2: "Stop Refreshing" QML Button Definition

The `onClicked` property is actually a Qt *slot*: whatever is connected to it gets executed as soon as the `clicked` signal fires from the button widget.

Users can perform one-line JavaScript method calls (like `onClicked: toggleText()`), or define short anonymous functions (like the above). JavaScript methods can be embedded in the object itself or in external source files.

7.2.4 JavaScript Host Environment

Although it can run JavaScript code, QML provides a JavaScript host environment tailored for writing QML applications^[83]. This environment is different from the host environment provided by a browser or a server JavaScript environment such as Node.js: for example, QML does not provide a `window` object or DOM API as commonly found in a browser environment, but it provides a global QML object that exposes functions such as `qsTr()` for internationalization, the `console` object, etc^[84]. It also applies a few additional restriction, such as the impossibility to modify the global object^[86].

Due to this fact, some commonly used functions and objects are not available in the QML engine. A complete list of the available ones can be found in the documentation^[85]. The issue in this case is related to the increased difficulties of finding JavaScript libraries that make use only of the available features. In addition, every JavaScript feature that is not in the ECMA-262 specification^[87] is not supported by QML.

Another point of concern is due to the fact that the QML engine provides optimization mostly for the so-called *short bindings*, simple assignments of values or single function calls to the backend, which are supposed to be the

most common use case for JavaScript in QML. However, the QML environment provides also tools for multithreading, like worker threads^[88] and similar features.

7.2.5 Charting

Since Qt5.6^[94], Qt includes by default what used to be an add-on, called QtCharts. It provides all the typical functionalities of a charting library, so different types of graphs, plotting, plot interactions, multi-series charts, and many more^[95]. The library, however, is unexpectedly bad developed.

As some experienced developers says on official Qt forums, "QtCharts is one of the worst developed modules of the entire Qt"^[98], and its porting into QtQuick just made everything worse. The QML module indeed features a great amount of negative aspects, namely:

1. *Inconsistent API*. An example: `LineSeries` (points are edges of a line) and `ScatterSeries` (points are dots on the chart) have a `.clear()` method to remove all the points. Instead, `AreaSeries` (same as a `LineSeries`, but filling the underlying area with color) does not. This happens because, for obscure reasons, an `AreaSeries` does not inherit from `XYSeries` as `LineSeries` and `ScatterSeries` do^[99;100].
2. *Counter-intuitive API*. An example: suppose the user wants to have the Y axis of a plot showing up on the right side, instead than on the left. Qt's `ValueAxis` have a property called `alignment`, which cannot be assigned. The chart main object, `ChartView`, has no reference to the position of the axis either. The correct way is to set a specific property, `axisYRight`, on *all the chart's series*. Funny enough, this property is exposed only if the series are declared in QML, and not if they are generated dynamically^[101]. In this last case, one must provide a dummy left axis that is automatically thrown away after the right axis has been set.
3. *Missing functionalities*. An example: on the most basic chart's axis type, `ValueAxis`, the position of ticks cannot be controlled. On the most advanced one, `CategoryAxis`, ticks must be added manually, and have to be added from the smaller to the larger. Any other operation fails silently without drawing the tick^[102]. In either case, the user has no

control whatsoever on the minor ticks of the grid, because `CategoryAxis` does not have the possibility to draw minor ticks at all.

The list of all these trivial issues is very long and gets longer and longer the more the users try to customize their plots in a way that is a little non standard, especially for moving plots.

Moreover, being QML a fairly new paradigm for Qt and still subject to heavy development, no other plotting libraries seems to be available to date.

7.3 The application



Figure 7.1: *QtQuick application has a very modern look and feel and can be styled to look native on most platforms, but shows poor charting performance.*

The application itself runs smoothly and has a very pleasant, modern look. All the target controls were implemented without big problems using a mix of QML and JavaScript.

One interesting aspect of this paradigm is that the GUI definition and its internal, small logic are decoupled, but not from a language perspective. While MVC is still being used also to wire up the GUI reactive behavior (up to a

Qt Quick: QML & JavaScript

certain degree), all the three layers of the architecture are written in QML and JavaScript together.

An example is the model. The application's model is made of two objects: a small `DataSource` object, that fetches the data from the backend to populate the graph, and a `ListObjects`, that simulates a larger model and populates the `ComboBoxes` and the `ListView`. Both of these classes are first declared in QML, and then they are extended with a few JavaScript methods to define their behavior. This allows us to make use of Qt's features and APIs from inside JavaScript, but also makes the code much clearer. Let's take `DataSource.qml` as an example:

```
1 Item {
2     signal dataReceived;
3     property var dataset: []
4     /* ... more properties ... */
5     function connect(){
6         dataReader.subscribe("LHC.BSRTS.5R4.B1/Image#imageSet", "",
7             "")
8     }
9     function disconnect(){
10        dataReader.unsubscribe("LHC.BSRTS.5R4.B1/Image#imageSet", "
11            ", "")
12    }
13    Rda3QtDPA {
14        id: dataReader
15        onRdaDataAcquiredChanged: {
16            /* ... more code ... */
17            dataReceived() // This call triggers the parent's signal
18        }
19    }
20 }
```

Listing 7.3: DataSource.qml

The above class does not have any graphic impact on the application: it is completely a Model class from an MVC point of view. However, it can still be defined in QML and this allows us, in the `onDataAcquiredChanged` slot, to emit a Qt signal that will be received by the graphical observer's and subsequently rendered.

Controller's methods like `.connect()` and `.disconnect()` are added to the class for convenience, but they can be easily moved into a dedicated controller class at need.

With respect to charting, we also got non satisfactory outcomes. By making use of the QtCharts library according to tutorials and its own APIs design,

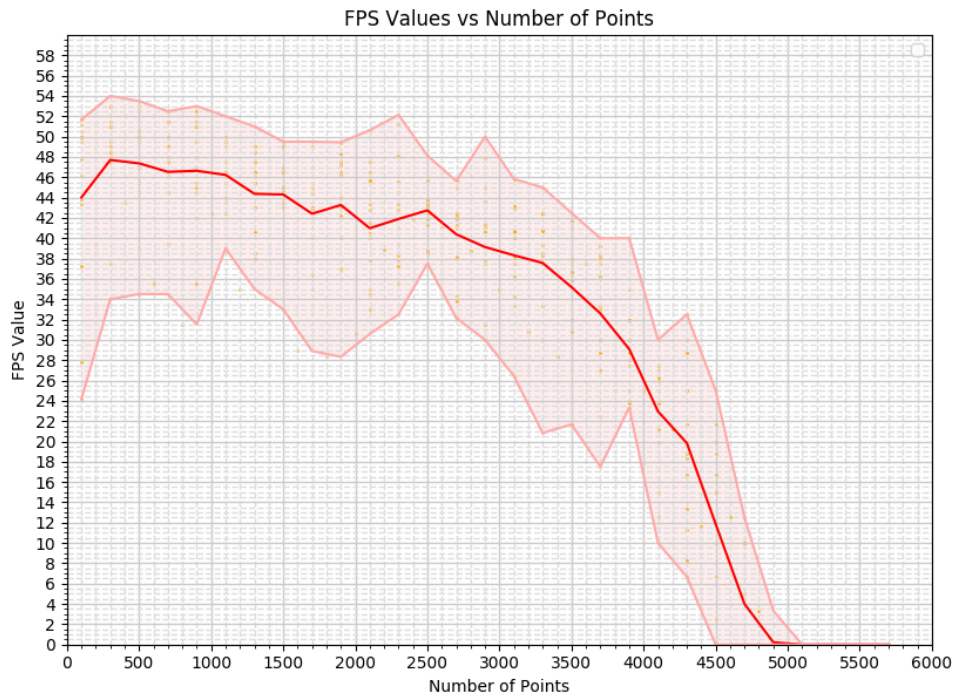


Figure 7.2: Charting performance of a QtQuick application. Due to the repainting issue, the frame rate drops quickly to zero once we try to render more than a few thousand points.

even after careful tuning and code reviews, the performance we obtained was quite poor: the application was not able to render many more point than a plain JavaFX application, so a few thousands of points only. However, in the documentation of QtCharts, we found that the library should be very powerful and able to render even a few hundred thousands points.

After careful inspection of the situation and some research, we eventually managed to identify the bottleneck while developing the Python version of the benchmark. The concept is that a QML application redraws the chart every time it is notified of a change from the LineSeries' underlying model. However, the APIs exposed to QML are not the full C++ APIs: this means that the only way to modify the LineSeries model is by calling the `.append(x, y)` method, that adds one single point. A `.clear()` method is also available, but direct access to the model is not possible, nor a `.replaceAll()` method exists.

In practice, this means that the LineSeries' model will emit a `pointAdded` signal at each append, and that the chart will be redrawn *at each append*.

This procedure clearly does not scale, but no other way to update the

model was found.

7.4 Tooling

To edit QML files and develop QtQuick application, Qt provides a custom IDE called QtCreator^[45]. In fact, QtCreator allows the developer both to produce QML files by hand or by drag-and-dropping widgets on a canvas, and to write C++ code by providing code completion capabilities, a debugger, a compiler, a QML profiler^[46], and similar features.

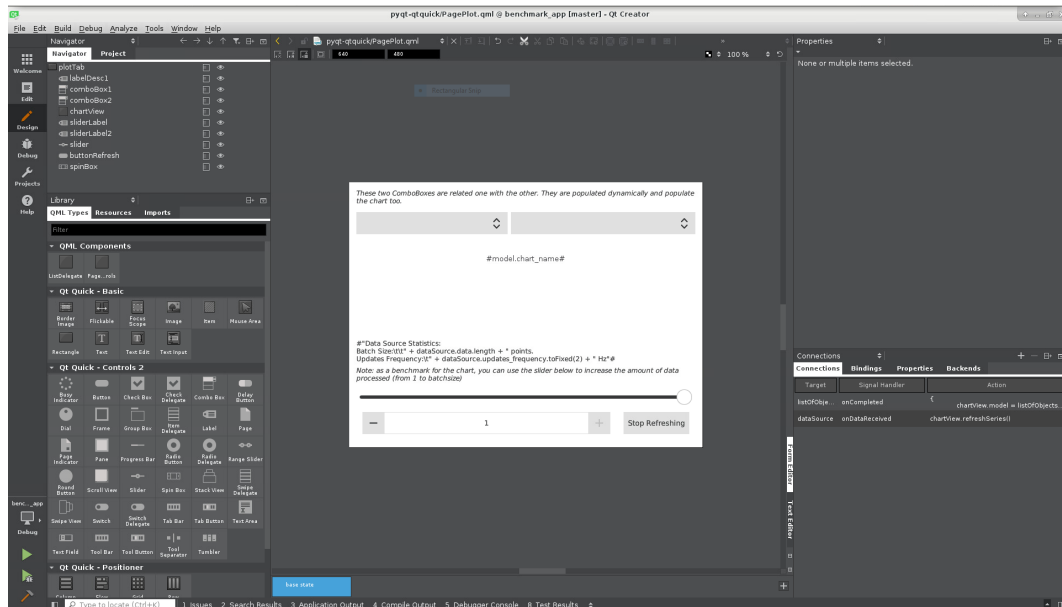


Figure 7.3: QtCreator's QML editing mode

While QtCreator may look superior to SceneBuilder with respect to the number of features offered, it is worth noticing that SceneBuilder could be easily integrated to Eclipse with a simple plugin, thus obtaining a very similar development environment than the one QtCreator offers. In addition, being Eclipse a general purpose IDE that can be further extended with many more plugins, the JavaFX solution may be even seen as preferable.

7.5 Documentation

Qt QML documentation is very complete, comprehensive and up-to-date. It spans from API description to tutorials for QtCreator, best-practices, and

7.6 A Side Attempt: TypeScript

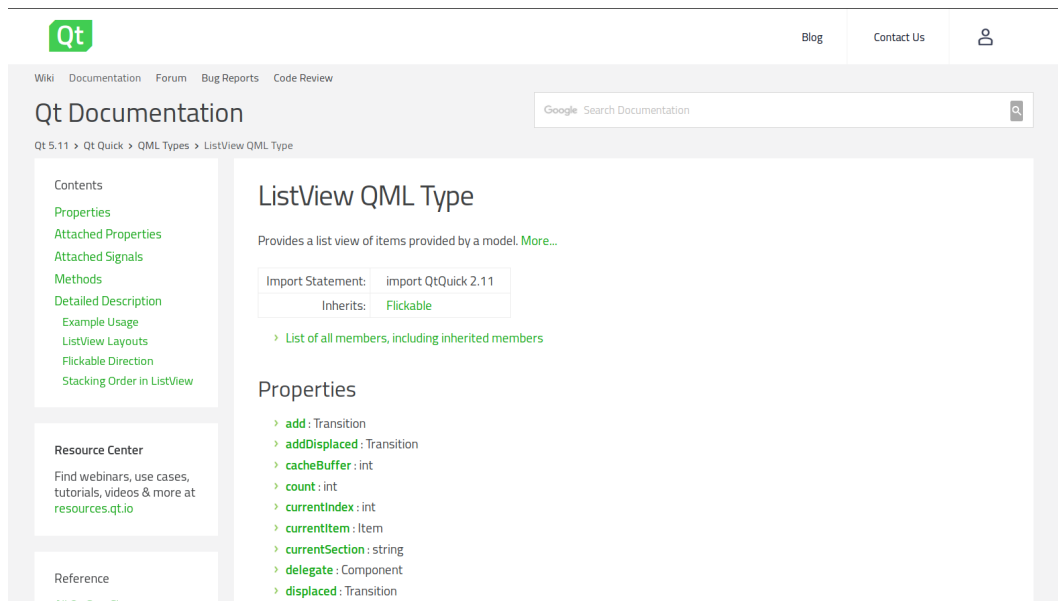


Figure 7.4: Sample entry from the Qt Documentation’s QML Types Description. One can notice from the Content’s side pane that the documentation tries to leave no detail uncovered.

features a very large quantity of examples and commented examples.

A Qt Wiki^[47] also exists, but is in general more outdated and less readable than the excellent official one. A huge amount of unofficial online tutorial and StackOverflow answers also helps a lot the development.

7.6 A Side Attempt: TypeScript

The most relevant issue with this concept are the traits of JavaScript itself. Being a scripting, dynamically-typed language makes it not very suitable to develop large-scale applications, but the main drawback is the difficulty of producing high-quality code with it, especially if the task is assigned to less experienced developers as our customers are.

A suitable alternative is TypeScript: an open-source programming language which is a strict syntactical superset of JavaScript and adds optional static typing to the language.

TypeScript is designed for development of large applications and transcompiles to JavaScript, therefore it can be run into the QML engine. The transcompiler, `tsc`, also allow different compilation options in order to adapt to different

JavaScript environment^[89], and therefore it can be easily adapted to generate QML-supported JavaScript code.

A small drawback of TypeScript, from this perspective, is that it accepts any JavaScript code as valid TypeScript code: therefore its restrictions, the strong typing and all the additional features can be easily eliminated by an indiscriminate usage of this backdoor. Possible solutions include some compiler settings that would make it stricter with the allowed JavaScript, but nonetheless, this potentially dangerous flaw remains.

7.6.1 Node.js Modules

The most common use case for TypeScript is server-side development. Indeed, one of the biggest frameworks supporting TypeScript is Node.js, an asynchronous event driven JavaScript runtime designed to build scalable network applications^[90].

Node's infrastructure relies on a huge number of small JavaScript and TypeScript packages, which provide the most diverse functionalities. The TypeScript compiler itself is usually installed via Node's own package manager, `npm`^[91].

An attempt to use such packages to provide additional functionalities to QML was therefore carried on. However, as said, QML JavaScript host environment is different from Node's, so the modules has to be ported first.

The idea was not new and we identified at least two projects that aimed at automating this complex task: `Brig`^[92] and `Quickly`^[93]. Both of them were inspected carefully and tested with different version of Node, Qt and `npm`, but they were both not working, not even with the suggested versions of the entire development stack. For `Quickly`, the banners in the GitHub repository itself signal the bad status of the codebase. The two projects were most likely abandoned.

7.6.2 Tooling

The tooling landscape for TypeScript is surprisingly good, even if far from the comfort of "standard" QML development. In fact QtCreator can be setup to compile TypeScript code into JavaScript before compiling C++ code, using

any `tsc` distribution available on the system, but it does not support TypeScript editing, not even with syntax highlighting. This makes it almost totally unsuitable as a development environment.

On the other hand, Visual Studio Code^[76] offers a very complete support for TypeScript (being both developed by the same company, Microsoft), and a partial support for QML as well, at least from the coding perspective. In fact, VSCode does not provide any WYSIWYG functionality for QML as QtCreator does.

7.6.3 Evaluation Outcomes

First of all, it should be noticed that the time invested in the TypeScript evaluation was considerably shorter than the one invested over the other options, mainly due to the increasing difficulties we were facing in the development itself while carrying it on.

In fact, despite the sufficient quality of the tooling compared with the standard Qt tools, other types of issues arised along the evaluation. Most of them were related to the difficulty of any IDE to match TypeScript's strict checks at compile time and the QML environment characteristics. An example will clarify the point.

The QML engine exposes to all scripts a `Qt` object, that can be used to communicate in some specific ways with the backend. For example, `Qt.createObject()` can be used to add new QML elements to the DOM. However, there is no way to set the TypeScript compiler to recognize `Qt` as a valid object exposed by the JavaScript host environment, neither through any IDE, neither from the command line. The compiler could be set to compile even in case of compiler errors, and the generated code was working just fine in the QML engine: however, such a setting totally vanishes the advantages of using TypeScript in place of JavaScript, because it would allow any broken code to run.

This issue, along with the impossibility of using most JavaScript and TypeScript libraries into the QML engine and the far from perfect quality of tooling, proved this option to be nonviable.

The door is still open, however, for the use of JavaScript as a communication tool between Java and Qt. Some other teams have scheduled the

development of interfaces with JavaScript for their services, so a future review of our evaluation might be scheduled for later, when our backend will offer something concrete to test on. In addition, there are rumors about a possible, future native support of TypeScript in QML^[48]. If such a scenario becomes more concrete, a re-evaluation of this approach may be considered.

7.7 Outcomes

QtQuick development is clearly the cutting-edge technology for GUI development within the Qt framework. The community is lively, many Qt-based projects are moving from older implementations to QML-based interfaces, and the Qt Company is investing the biggest part of its effort into its further development. The community is very lively and active, the documentation is in pristine conditions, and a great deal of material is available for learning.

What raises concerns toward its adoption is the overall feeling of QtQuick being still too young, more suitable for early adopters and small-scale projects. Despite the small hype surrounding it, it may be not yet suitable for the very stable, decades-stable environment of CERN's control system's infrastructure.

Other than the poor charting performances achievable in QML, one clear indicator is the lack of some basic widgets in the latest widget library for QML. Basically everything that does not have a flat model, namely TableViews and TreeViews, are still missing from the standard widget library, QtQuick Controls 2. Albeit available in the QtQuick Controls 1 version, they provide such a different look with respect to the following generation that they cannot be mixed successfully without the interface looking very awkward.

While they will surely be available very soon, it is easy to see how the technology feels still somehow immature for our use case.

The final decision with regard to QtQuick was driven mainly by this feeling: we considered QML still too young to be eligible as a recommended GUI framework, but it will be possible, for early adopters and more experienced users, to try it out in different ways and start building an ecosystem within CERN around QtQuick development.

Chapter 8

Qt over Python: PyQt5

Nowadays Python is one of the most popular programming languages available, and arguably the most versatile, easy to learn and to use. While born for prototyping purposes, its usage expanded way further its original scope, being now one of the most used languages used by the scientific community for research, data analysis, and so on.

However, Python is a scripting language, which makes it almost unsuitable for large production-level software that focuses on performance. The need for a strong graphic library is not really felt in the Python community and, apart from very specific charting libraries such as matplotlib or seaborn, no native, complete Python GUI libraries has ever been developed. Indeed, being Python so versatile, it was considered easier to develop bindings with already existing graphic libraries, and therefore to leverage the performance of native code while retaining the ease of use of Python.

Different attempts in this directions has been made with respect of Qt, and most of them eventually converged into the same set of extremely similar APIs. Examples of these bindings are PySide, PyQt, QtPy, PyQt5 and the newer PySide2. Our focus was put on the one which looked, at a first sight, the most healthy and stable: PyQt5^[53].

PyQt5 is a very complete binding over Qt APIs. This means that PyQt5 can be used to produce QtQuick and QtWidgets applications alike, just as one would do with C++. In this case, both approaches were evaluated.

8.1 Installation and setup

The setup process of PyQt5 is no different than for any other Python package, and shares the strengths and the weaknesses of Python dependency management.

As for most Python packages, there are different ways to install PyQt5:

- *With Anaconda^[50]*. PyQt5 can be installed through the `conda` package manager and in general comes with the default installation of Anaconda. However, the latest version available with Anaconda is not up-to-date, and also misses some Qt modules, namely the standard charting library.
- *With pip^[51]*. PyPi also hosts a precompiled PyQt5 distribution, which instead is up-to-date with the latest PyQt5 release and lacks no modules.
- *From source^[52]*. The library has the binding module, `sip`, as only dependency: therefore, building from source is feasible. The `sip` package can be also installed from source.

As for plain Qt, in case of adoption we plan to make PyQt5 and all the tools already available for the users on all the machines.

8.2 Development process

PyQt5 is a very tiny wrapper on top of the C++ APIs of Qt. Every feature of the original APIs has been ported to Python in the closest fashion possible, making PyQt5 usage extremely close to Qt native coding. In most cases, the same line of code runs in the exactly same way in a C++ class and in a Python one (at least from a developer's perspective).

In order to assess in practice how trivial is to port code from C++ to Python, it is interesting to directly compare a few lines of code.

8.2.1 Code Comparison

The snippet below perform an ordinary QtQuick application startup: instantiate `QGuiApplication` objects, loads the QML file that models the interface, exposes a few classes and instances from the backend and eventually

launches the application.

```

1 #include <QGuiApplication>
2 #include <QQmlApplicationEngine>
3 #include <QQmlEngine>
4 #include <QQmlContext>
5 #include "custom_model.h"
6
7 int main(int argc, char *argv[])
8 {
9     // Instantiate the application
10    QGuiApplication app(argc, argv);
11    QQmlApplicationEngine engine;
12
13    // Load the QML file
14    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
15    if (engine.rootObjects().isEmpty())
16        return -1;
17
18    // Instantiate the model
19    ListOfObjects list_of_objects;
20    DataSource data_source;
21
22    // Expose C++ objects to QML
23    engine.rootContext()->setContextProperty("listOfObjects",
24        list_of_objects);
25    engine.rootContext()->setContextProperty("dataSource",
26        data_source);
27
28    // Start the application
29    return app.exec();
30 }

```

Listing 8.1: Qt C++ Original Code

```

1 # -*- coding: utf-8 -*-
2 import sys, os
3 from custom_model import ListOfObjects, DataSource
4
5 from PyQt5.QtGui import QGuiApplication
6 from PyQt5.QtQml import QQmlApplicationEngine
7 from PyQt5.QtCore import QUrl
8
9 def main():
10
11    # Instantiate the application and the engine
12    app = QGuiApplication(sys.argv)
13    engine = QQmlApplicationEngine(parent=app)
14
15    # Load the QML file
16    engine.load(QUrl('qrc:/main.qml'))
17    if engine.rootObjects().isEmpty():
18        return -1;
19
20    # Instantiate the model
21    list_of_objects = ListOfObjects();

```

Qt over Python: PyQt5

```
22 data_source = DataSource();
23
24 # Expose Python objects to QML
25 engine.rootContext().setContextProperty("listOfObjects",
    list_of_objects);
26 engine.rootContext().setContextProperty("dataSource",
    data_source);
27
28 # Start the application
29 app.exec_()
30
31 if __name__ == "__main__": main()
```

Listing 8.2: PyQt5 Python Code

The similarity between the two should be now evident. Every single line of the C++ code can be matched with its Python counterpart and, apart from very language specific features like the imports and the `main` syntax, the code itself is basically identical.

This binding style has its pros and cons. On the positive side, it makes the porting from one language to the other very straightforward, almost trivial; on the negative side, it produces APIs which sometimes look very unfamiliar for Python developers, because they reflect C++ coding styles.

```
# Python:
Point point = new Point(1.0, 3.0)
QMetaObject.invokeMethod(self._series, "append", Qt.AutoConnection,
    Q_ARG('double', point.x() ),
    Q_ARG('double', point.y() )
)

// C++
Point point(1.0, 3.0);
QMetaObject::invokeMethod(m_Series, "append", Qt::AutoConnection,
    Q_ARG(double, point.x()),
    Q_ARG(double, point.y()),
);
```

Figure 8.1: The Meta Object System's APIs, as an example of how the Python code can get to look very unfamiliar to Python developers.

8.2.2 QtQuick

One positive aspect found during the implementation of the PyQt5 QtQuick benchmark app was that most of the QML could be reused without any mod-

ification. Clearly only the code defining the graphical interface was reused, while the model has been moved back to Python.

This achievement clearly exemplifies how decoupled QtQuick applications are and how well the MVC pattern is implemented in this architecture.

8.2.3 QtWidgets

The development of the Qt Widgets version of the benchmark also carried good news. In this case, the reuse of code was two-sided:

1. The XML could be reused from the C++ version with minor modifications. The only real difference was to setup the editor to get the `.ui` file compiled through `pyuic5` instead of `uic`, in order to generate a Python class and not a C++ one.
2. The model could be reused from the QtQuick implementation. In this case a little logic had to be moved back from the interface (due to the lack of JavaScript support from `.ui` files), but the modifications were extremely limited and did not made the model unusable by QML.

Trying to keep the same model for both applications showed how much responsibility the developers have in the Qt Widgets approach: the model can be completely decoupled, as in QML, but the decoupling is not enforced in any way.

8.3 The application

8.3.1 QtQuick

The resulting application shows not defect. It runs easily the exact same declarative code used for the pure QtQuick application previously developed, calling Python methods instead of JavaScript ones.

It provides also a lot of additional advantages, due to the possibility of leveraging Qt classes through their wrapped C++ API. To give an idea of the difference, we can have a look at the charting performance.

As we described in the QtQuick chapter, a "naive" PyQt5 QML application would simply react to a signal from the backend by triggering a repaint of

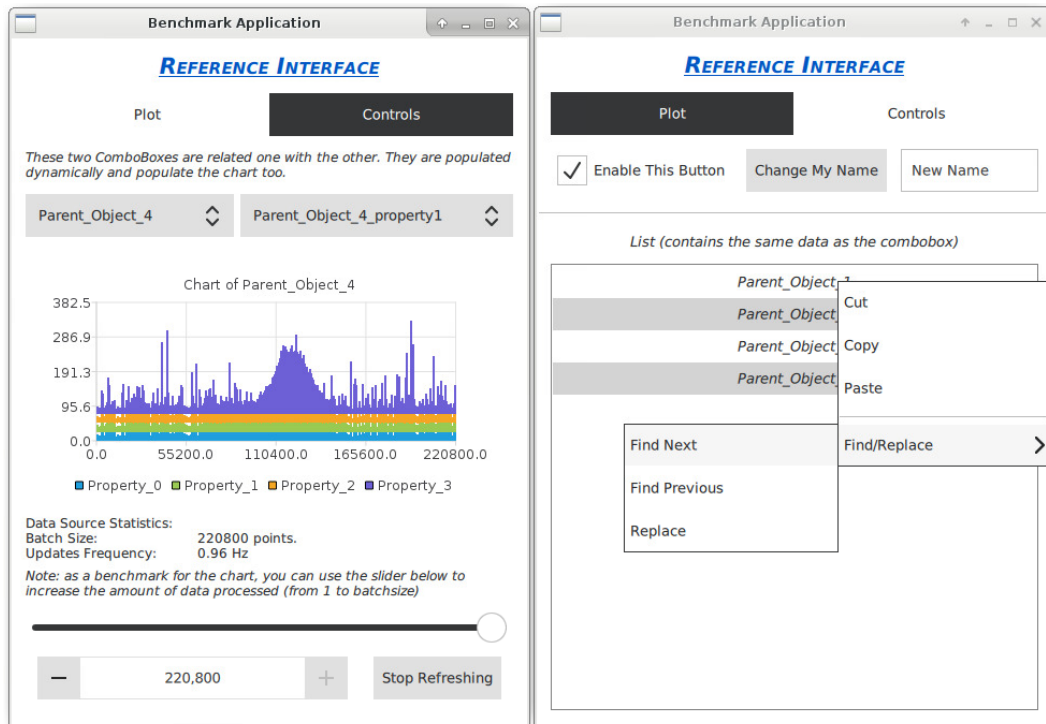


Figure 8.2: With small workarounds and leveraging the native Qt APIs, PyQt5 QtQuick applications can reach impressive charting performance.

the entire plot for every append operation performed on the QML LineSeries object. This behavior throttles down wildly the capabilities of the widget. In practice, such an application can render a maximum of approx. 1000 points per update, which is far from ideal.

However, the Python backend provides a workaround. If the reference to the QML LineSeries is passed back to the Python script, namely at startup, it will expose some methods that were hidden to QML. Among those lies the extremely useful `.blockSignals()` methods, inherited from `QObject`.

The `.blockSignals()` method is exactly what is needed to remove the repainting obstacle. Indeed, one can temporary disconnect the model of the widget from its graphical part, perform any modification, plug back the two components, and then fire a generic signal to trigger the repaint. By performing this operation, the repainting overhead is completely removed and the widget performances increase dramatically: in fact, it becomes capable of plotting the entire data source while its size stays near the 200 000 points.

8.3 The application

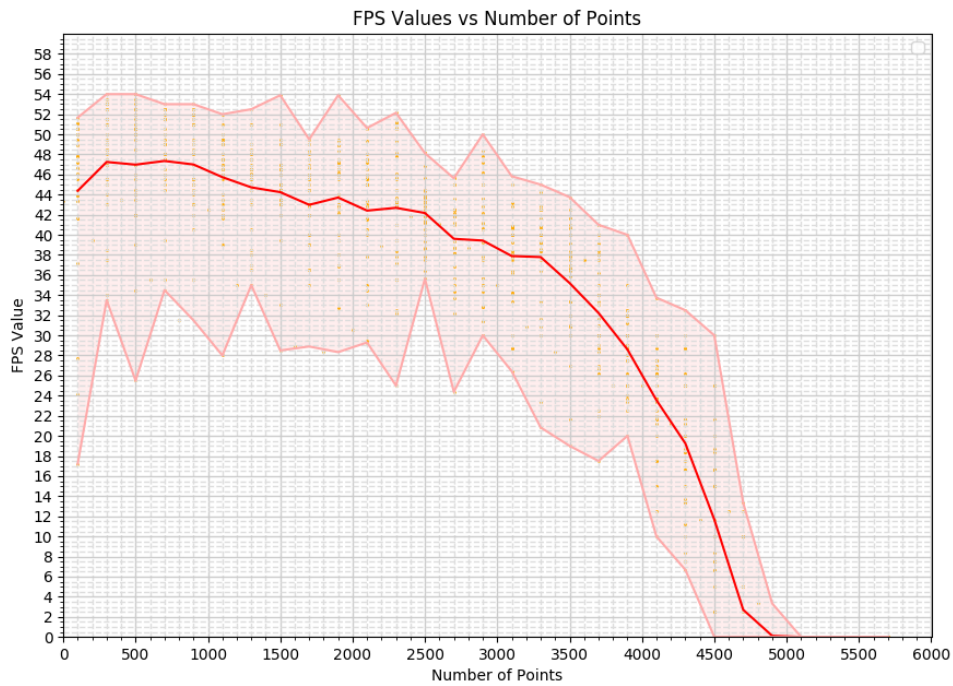


Figure 8.3: Charting performance of a naive PyQt5 QtQuick application. As we can see, the FPS count drops quickly to zero once we try to render more than a few thousand points.

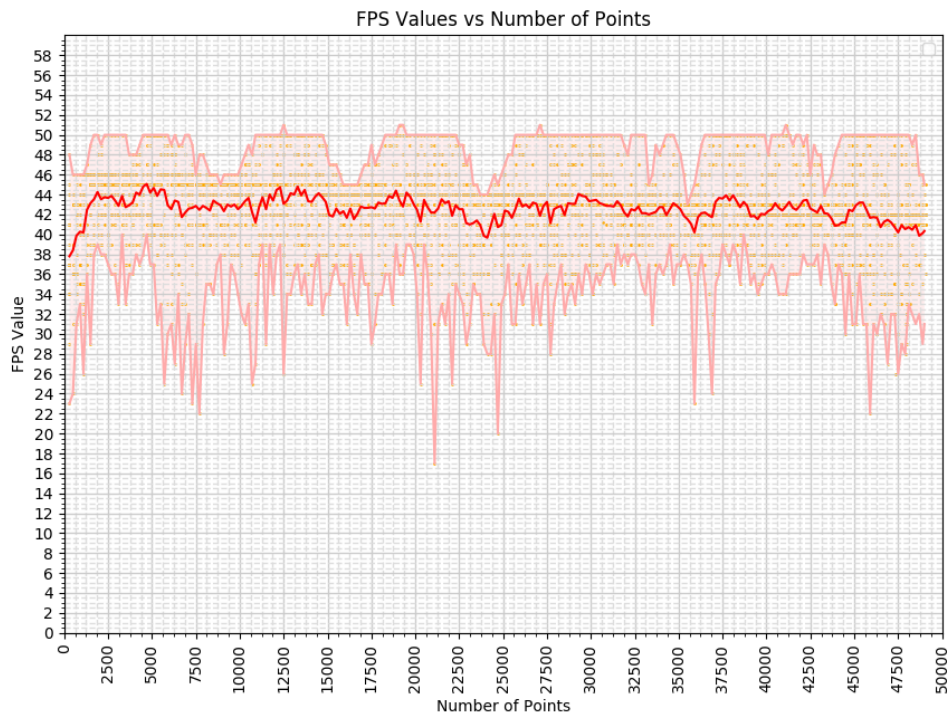


Figure 8.4: Charting performance of a non-naive PyQt5 QtQuick application. As we can see, the frames-per-second count keeps more or less stable with the size of the series plotted.

A Side Attempt: Matplotlib integration

One of the most relevant points in favor of the adoption of Python for GUI development is its common adoption into the scientific community. Most of actual developers of monitoring interfaces for the accelerators are, indeed, physicists and technicians: people whose specialization is completely unrelated to software engineering, who need some easy to use solution in order to focus on the actual business logic of the applications.

In order to take full advantage of this point we should consider how PyQt5, being a thin wrapper over a C++ library feels somehow different from the traditional Python coding the operators are used to. To cope with this gap, we evaluated the possibility to develop PyQt5 QtQuick applications embedding other widespread Python plotting libraries, namely Matplotlib^[68] and Seaborn^[69].

Attempts to perform such integration were already present in two different forms:

1. A small proof of concept about the actual possibility of integrating a matplotlib chart into QML^[70].
2. A working integration module between matplotlib and QtWidgets through `.ui` files, already tested by some operators in their applications^[71].

The target included both the possibility of painting a chart using the standard Matplotlib APIs, but also to interact with it on the frontend in the same way as a standard Matplotlib plot would: that included support for panning, zooming, history of moves, the possibility to save the chart as an image, and a few layout settings.

Given these inputs, the integration was performed successfully. The code of the proof of concept was refactored into a standalone Python module, `pip`-installable^[72;73], exposing the same APIs and features of the older XML-based integration.

On top of this successful result, additional test were performed to check whether Seaborn plots could be integrated into a PyQt5 frontend in the same way. The investigation showed that the same source code can easily render Seaborn plots in the same way it renders matplotlib's.

8.3 The application



Figure 8.5: Matplotlib and Seaborn plots embedded into a PyQt5 QML frontend

This specific integration was not tested on performance. However, matplotlib is not meant to be used for fast updating charts or very massive datasets, but more likely for static or slow updating figures.

8.3.2 QtWidgets

Also the QtWidgets application can be developed without big effort. The code is extremely similar to its C++ implementation, as seen before in the Code Comparison section, and a lot of code from other implementations could be reused with very minor modifications (probably not necessary and due to my own inexperience).

In this context, however, manipulating the objects from the Python side is even more natural than in QML. Indeed in the QtQuick implementation the user had to pass back the reference to the `LineSeries` in order to be able to call the `.blockSignals()` function; in Qt Widgets instead the reference is readily available.

Qt over Python: PyQt5

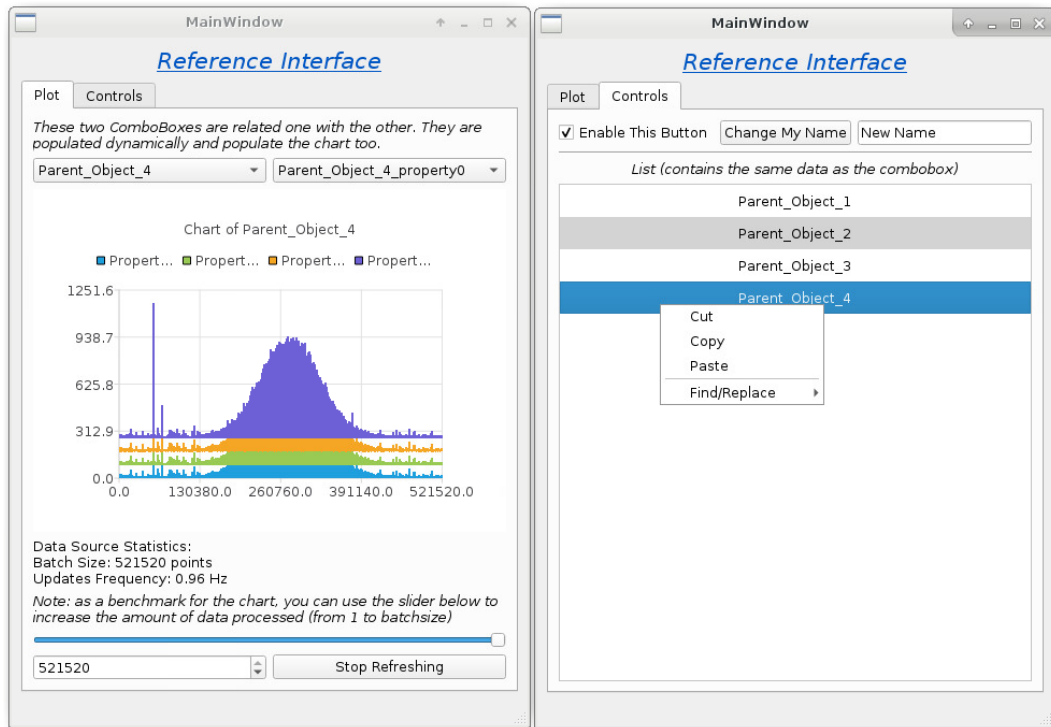


Figure 8.6: Also a PyQt5 QtWidgets application can achieve impressive charting performance.

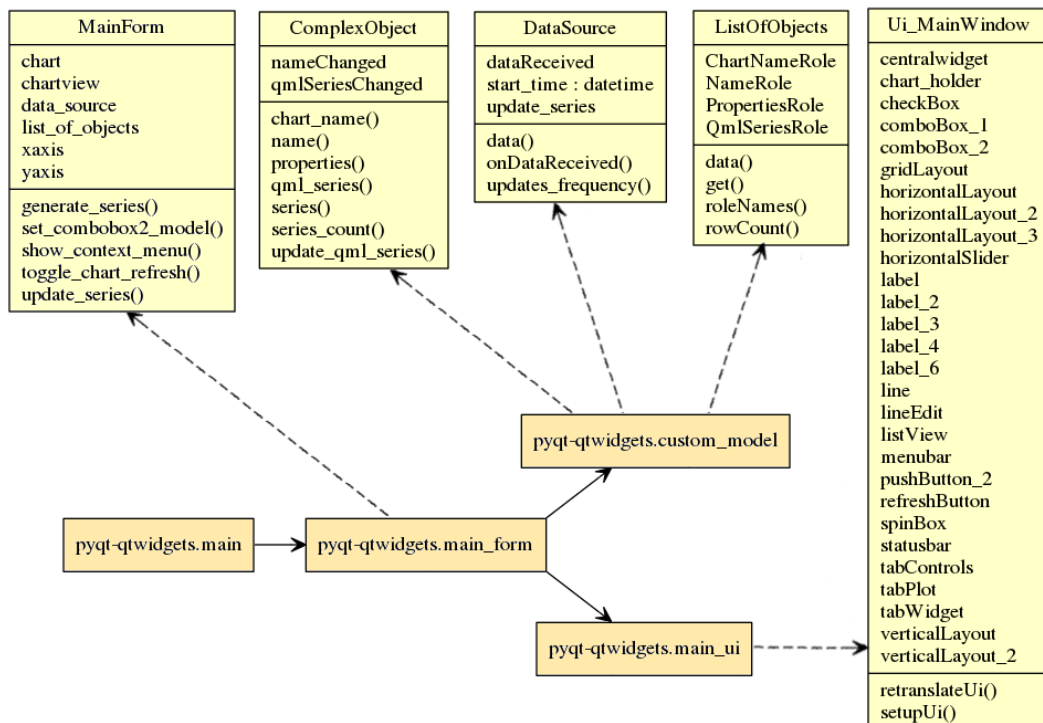


Figure 8.7: Class Diagram for the PyQt5 QtWidgets application. Darker boxes represent source files, lighter boxes are the classes. Solid arrows are import statements, dashed arrows represent inclusion.

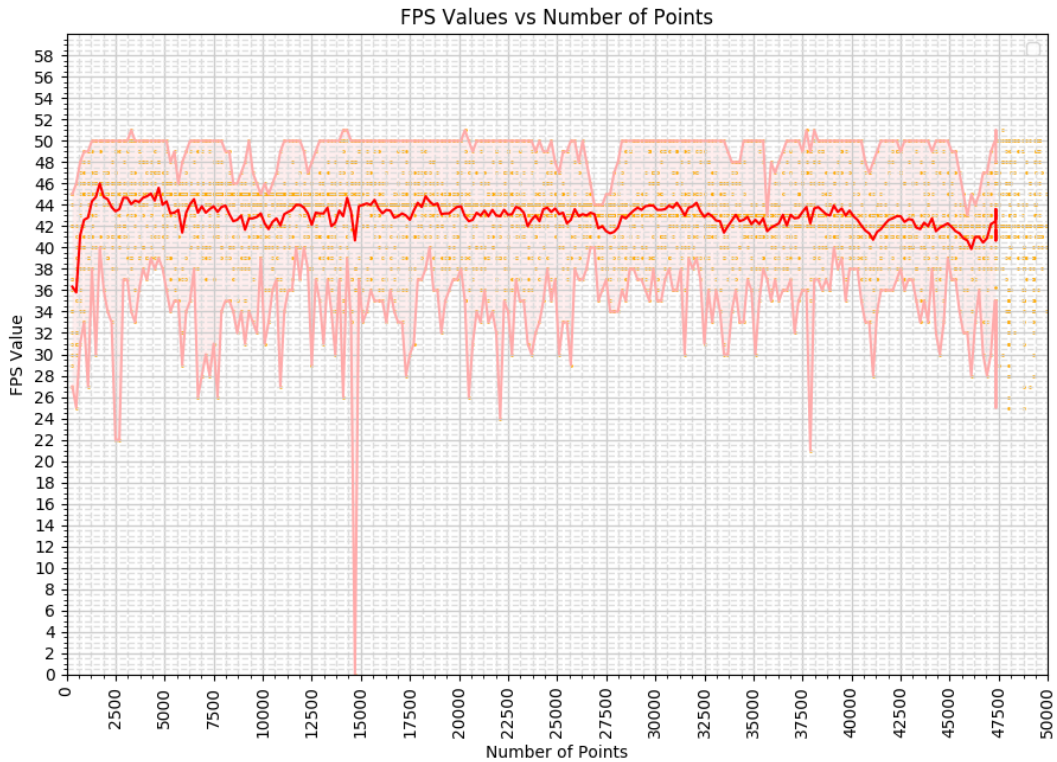


Figure 8.8: Charting performance of a PyQt5 QtWidgets application. Like in the QtQuick case, the frame rate count keeps more or less stable with the size of the series plotted. This is also the performance expected for a C++ Qt Widgets application, that we could not assess.

As we can see on the screenshots, the look of the application is very different from a QtQuick one, more native and desktop-like.

8.4 Tooling

8.4.1 QtQuick

With regard of tooling, and with respect to the average tooling quality of Qt products, PyQt5 for QML does not shine. Being a community based library, PyQt5 does not benefit from official support from the Qt Company, which maintains Qt Creator. Therefore, Qt Creator offers minimal Python support, which basically consists in syntax-highlighting only. Environments management, package management, and even code completion and linting are totally unsupported and left to the user.

For the scope of this evaluation, other general purpose IDEs were used, namely PyCharm^[75], VSCode^[76], and even bare text editors like Gedit^[77]. In most cases, IDEs providing full Python support were not providing any support for QML, so making necessary for developers to use two IDEs in parallel, which is far from ideal. An exception in this sense is VSCode, which features a wide degree of features and in general a comfortable development environment for both Python and QML, once all the required plugins^[78;79] are correctly set up. However, it cannot provide anything close to Qt Creator's Design Mode for QML files.

8.4.2 QtWidgets

For Qt Designer the situation is way less critical than it is for QtQuick. Indeed Qt Designer has a number of plugins that makes Python development pretty much straightforward.

What is truly needed in this cases is not full Python support, as Qt Designer is not meant to be an IDE, but simply an `.ui` files editor. The only configuration required is to make Qt Designer compile the `.ui` file by means of the Python User Interface Compiler, `pyuic`, instead of the standard `uic`.

Once the tools are setup, Qt Designer can be used just it would for plain C++ development. In case the user wants to embed some custom PyQt5 widgets exposed by some library, a plugin has been developed to ease the process of importing them into Qt Designer, making the process almost trivial^[66].

In addition, Qt Designer can be linked to a proper Python IDE, like PyCharm, as default external editor for `.ui` files, creating a development environment very close to the JavaFX & SceneBuilder pair.

8.5 Documentation

Being so close to native Qt coding, PyQt5 can directly leverage the completeness of original Qt documentation. In most cases, the same line of code runs in the exactly same way in a C++ class and in a Python one, and very often the PyQt5 documentation simply links the official, C++ Qt one, without adding a word to it.

[PyQt 5.8.2 Reference Guide](#) » [PyQt5 Class Reference](#) » [previous](#) | [next](#) | [modules](#) | [index](#)



QColor

`class PyQt5.QtGui.QColor`
[C++ documentation](#)

Previous topic

[QCollatorSortKey](#)

Next topic

[QColorDialog](#)

Quick search

[PyQt 5.8.2 Reference Guide](#) » [PyQt5 Class Reference](#) » [previous](#) | [next](#) | [modules](#) | [index](#)

© Copyright 2015 Riverbank Computing Limited. Created using [Sphinx](#) 1.5.3.

Figure 8.9: Example page from the PyQt5 Reference Guide. Most of the documentation looks like this.

In addition to API documentation Riverbank^[67], the company currently providing support for PyQt5, provides also documentation for specific Python structures, as well as tutorials, how-to-s, and code conventions for both QtQuick and QtWidgets, all of these usually fully up-to-date and very useful to developers.

8.6 Outcomes

The outcomes of the PyQt5 evaluation are generally positive. It proved to be very solid and complete, supporting basically all its features seamlessly and therefore qualifying well as a candidate primary GUI framework for our use case.

The situation is not optimal, especially under the tooling perspective, but many other positive aspects of this technology balance for it, like the ease of use of Python and PyQt5 itself, the possibility to leverage Qt5's own documentation, the lively community, and many more.

In addition to all of this, during our evaluation the Qt Company published an announcement about the release of a new Python binding, called Qt for

Qt over Python: PyQt5

Python, or PySide2^[80]. The release of a new Python binding for Qt may bring a lot of benefits, especially with regard to better tooling (the Qt Company maintains QtCreator, so Python support is likely to be implemented in the near future) and more guarantees about the stability of the binding itself. With regard of the actual usage, though, no substantial differences are foreseen between the two libraries, so an eventual porting of PyQt5 code to PySide2 would be trivial, if not automatic.

Adopting Python as main GUI development language, however, would require a substantial effort from the backend developers to offer a suitable interface for communication with Java. Indeed, plans for Python support from the backend were already started before the beginning of this evaluation and, even if they are still a long-term goal for the Section, they make the PyQt5 solution viable.

Chapter 9

Conclusions

At the end of this preliminary, broad evaluation of Qt, the results are somehow concerning, but not critically negative.

Many aspects of it were considered, many ways to make use of the toolkit were evaluated, and the main feeling we obtained from the exploration is of a very solid library, well developed, well supported, with a long heritage and a safe guarantee of being supported for many years to come. The real concern, indeed, lies only in the connection point between this excellent C++ library and the huge Java backend we need to provide GUI solutions for.

Most of the identified paths carry some big overhead or threat that must clearly be dealt with. Most of them look like "workarounds" for the lack of a direct binding between C++ and Java that was the dead QtJambi. The QtQuick solution might work in case of "weak" bindings (for example, through web APIs), but it would probably push the QML engine to the limits of its capabilities and bring us a brand new set of unknowns, due to performances, optimizations, customization, and so on. The Python solution, the most viable to date, looks like a good compromise, but still far from an optimal one: it would unnecessarily add a third language to an already complex stack of technologies, making maintenance, deployment and monitoring even more complex than how it is now.

The results of this evaluation clearly are not satisfactory enough to drive our commitment to Qt as main GUI framework for CERN just yet, but opens up some possibilities. A further investigation into other technologies, for example Web, will definitely be carried on and compared with the results of this evaluation before taking a final decision.

Conclusions

	Enforces Clear Architecture	Widgets Availability	Charting Performance	Tooling Quality	Documentation Quality	Can be integrated with Existing Systems	Ease of Development for Clients	Ease of Maintenance for Us	Health Status
JavaFX	✓	✓	✓	✓	✓	✓	✓	✓	-
QtJambi	-	✗	/	✓	✓	-	✓	✗	✗
QtQuick	-	-	✗	✓	✓	-	✓	-	✓
QtWidgets	-	✓	/	✓	✓	-	✗	✗	✓
PyQt5 QtQuick	✓	-	✓	-	✓	-	✓	-	✓
PyQt5 QtWidgets	-	✓	✓	-	✓	-	✓	-	✓

Figure 9.1: A qualitative comparison table to sum up the results of the evaluation. The colors stands for:

- Tick: Satisfactory or Overly Satisfactory
- Dash: Almost Satisfactory
- Cross: Non Satisfactory
- Slash: Not Assessed

For a detailed justification of the cells, please refer to the Annexes.

Appendices

Appendix A

Acronyms Definition

In this section all the relevant acronyms and some technical terms are listed and explained.

A.1 Definitions

GUI : stands for Graphic User Interface. The graphic component of an application that allows users to interact with the software through interaction with its graphic elements (buttons, icons, text fields...)

API : stands for Application Programming Interface. Identifies the set of methods a library makes available to the end user (a developer) in order to provide some functionalities.

FPS : stands for Frames Per Second, a measure for applications frame rate.

SDK : stands for Software Development Kit. It is a set of software development tools that allows the creation of applications for a certain software or platform.^[121]

JDK : stands for Java Development Kit.

AWT : stands for Abstract Windowing Toolkit. The first and oldest Java GUI development framework.^[106;107]

Swing : A Java GUI development framework.^[108;109]

Acronyms Definition

JavaFX : The latest GUI development framework developed by Oracle, meant to be the official successor of Swing.^[110]

SWT : stands for Standard Widget Toolkit, the GUI framework used by Eclipse.^[111]

GWT : stands for Google Widget Toolkit, the GUI framework used in Android.^[112]

JOGL : stands for Java OpenGL, a binding to Java of the OpenGL library.^[113]

CI : stands for Continuous Integration.^[114;115]

Qt : a cross-platform GUI framework used in automotive systems, with bindings to many programming languages^[7].

MVC : Stands for Model-View-Controller. Software architectural pattern that dictates the separation of the three main conceptual elements of a graphic application: the *model*, that represents the data, the *view*, that represents the bare graphic elements, and the *controller*, that models the logic connecting the graphic representation of the data to its underlying model^[24].

POJO : Stands for Plain Old Java Object. Refers to any Java class, especially one that does not belong to any specific framework.

WYSIWYG : Stands for "What you see is what you get". Typically said of tools that autogenerate code or markup basing on operations the user makes on the preview of the result^[116].

IDE : Stands for Integrated Development Environment. Said of softwares that aids the developers by providing tools like source code editor, build automation tools, debuggers, code completion, compilers, etc. Examples of such software are Eclipse, NetBeans, PyCharm...

SWIG : SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages, including Java^[118].

Appendix B

A JavaX Hello World Application

JavaFX offers two main patterns to implement GUIs: one that enforces the MVC pattern, and one that does not. In order to understand better the difference among these two methods, let's see how the same, minimal application is implemented according to the two strategies.

The application is extremely simplified and consist simply in a window with a button inside it. On click, the button prints "Hello World!" on the standard output.



Figure B.1: Minimal JavaFX application

B.1 First Approach: Pure Java

In this case, one single class is needed. It comprises all the elements of an MVC application, and gains in brevity what loses in decoupling^[119].

A JavaX Hello World Application

```
1 package helloworld;
2
3 import javafx.application.Application;
4 import javafx.event.ActionEvent;
5 import javafx.event.EventHandler;
6 import javafx.scene.Scene;
7 import javafx.scene.control.Button;
8 import javafx.scene.layout.StackPane;
9 import javafx.stage.Stage;
10
11 public class HelloWorld extends Application {
12
13     public static void main(String[] args) {
14         launch(args);
15     }
16
17     @Override
18     public void start(Stage primaryStage) {
19         // Set up the button
20         Button btn = new Button();
21         btn.setText("Say 'Hello World'");
22         btn.setOnAction(new EventHandler<ActionEvent>() {
23             @Override
24             public void handle(ActionEvent event) {
25                 System.out.println("Hello World!");
26             }
27         });
28         // Set up the window
29         primaryStage.setTitle("Hello World!");
30         StackPane root = new StackPane();
31         primaryStage.setScene(new Scene(root, 250, 100));
32         // Add the button to the scene
33         root.getChildren().add(btn);
34         // Show the window
35         primaryStage.show();
36     }
37 }
```

Listing B.1: 'HelloWorld.java'

The most relevant points are:

- The class has to extend `javafx.application.Application`
- The entry point is `launch(args)`, the only function call hosted in the main method.
- `launch(args)` will eventually call `start(Stage primaryStage)`, the "logic" entry point from the developer's perspective. It receives in input a `Stage`, which basically represents the window created by `launch(args)`. More information on this can be found in the official documentation^[22].

B.2 Hello World, MVC enforced

- `start(Stage primaryStage)` instantiates the interface one element at a time and set explicitly their parenting relationship into the Scene Graph.
- Custom Java calls to be issued in reaction to an event are described through the `EventHandler<ActionEvent>` class, by overriding its `handle()` method.

Although easy to understand, we can see how View and Controller are mixed together not even in a single class, but in a single method.

B.2 Hello World, MVC enforced

In this case we need three elements in order to build the same application: a View, a Controller, and an entry point^[120]. A Model is not needed only because the example is trivial.

```
1 package helloworld;
2
3 import javafx.application.Application;
4 import javafx.stage.Stage;
5 import javafx.scene.Scene;
6 import javafx.scene.layout.BorderPane;
7
8 public class MainApp extends Application {
9     @Override
10    public void start(Stage primaryStage) {
11        try {
12            // Load the FXML file
13            FXMLLoader loader = new FXMLLoader();
14            loader.setLocation(Main.class.getResource("Gui.fxml"));
15            BorderPane root = (BorderPane) loader.load();
16            // Setup the scene and shows the window
17            Scene scene = new Scene(root,400,400);
18            primaryStage.setScene(scene);
19            primaryStage.show();
20        } catch(Exception e) {
21            e.printStackTrace();
22        }
23    }
24    public static void main(String[] args) {
25        launch(args);
26    }
27 }
```

Listing B.2: 'MainApp.java'

A JavaX Hello World Application

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.control.Button?>
4 <?import javafx.scene.layout.BorderPane?>
5
6 <BorderPane maxHeight="-Infinity" maxWidth="-Infinity"
7     minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0"
8     prefWidth="400.0" xmlns="http://javafx.com/javafx/8.0.141"
9     xmlns:fx="http://javafx.com/fxml/1" fx:controller="
10 application.Controller">
    <center>
        <Button fx:id="clickMe" mnemonicParsing="false" onAction="#
            clickHandler" text="Say: &quot;Hello World!&quot;&quot;"
            BorderPane.alignment="CENTER" />
    </center>
</BorderPane>
```

Listing B.3: Gui.fxml

```
1 package helloworld;
2
3 import javafx.fxml.FXML;
4 import javafx.scene.control.Button;
5
6 public class Controller {
7
8     @FXML
9     private Button clickMe;
10
11     public Controller() {}
12
13     @FXML
14     private void clickHandler() {
15         System.out.println("Hello World!");
16     }
17 }
```

Listing B.4: Controller.java

Note how we don't even need the reference to the `clickMe` button, because the connection is done in the FXML file.

Even though such an architecture is an overkill for such a small example, the pattern scales up very well compared with the previous one, because responsibilities are clearly separated. In addition, the FXML file does not have to be handwritten, but is generated automatically by the SceneBuilder.

Appendix C

QtJambi Deployment Layout

In order for the reader to better understand the complexity of deploying QtJambi correctly, this section outlines the final deployed layout of our working QtJambi installation.

Most of the .dll and subfolder locations had to be guessed by cross-checking the documentation, StackOverflow and the folder structure of Qt itself.

What follows is a shortened version of the output of the `tree /f/a` command in the root directory (`qtjambi`). The entire output is several pages long (456 lines), therefore the content of the innermost folders has been trimmed. The shortened version however should give an idea of the size and complexity of the framework.

```
1 qtjambi
2 |
3 |   libEGL.dll
4 |   libGLESv2.dll
5 |   org_qtjambi_qt_concurrent5.dll
6 |   org_qtjambi_qt_core5.dll
7 |   org_qtjambi_qt_dbus5.dll
8 |   org_qtjambi_qt_gui5.dll
9 |   org_qtjambi_qt_help5.dll
10 |  org_qtjambi_qt_multimedia5.dll
11 |  org_qtjambi_qt_network5.dll
12 |  org_qtjambi_qt_opengl5.dll
13 |  org_qtjambi_qt_printsupport5.dll
14 |  org_qtjambi_qt_qml5.dll
15 |  org_qtjambi_qt_quick5.dll
16 |  org_qtjambi_qt_quick_widgets5.dll
17 |  org_qtjambi_qt_script5.dll
18 |  org_qtjambi_qt_scripttools5.dll
19 |  org_qtjambi_qt_sql5.dll
20 |  org_qtjambi_qt_svg5.dll
21 |  org_qtjambi_qt_test5.dll
22 |  org_qtjambi_qt_widgets5.dll
```

QtJambi Deployment Layout

```
23 | org_qtjambi_qt_xml5.dll
24 | org_qtjambi_tools_designer5.dll
25 | Qt5CLucene.dll
26 | Qt5Concurrent.dll
27 | Qt5Core.dll
28 | Qt5DBus.dll
29 | Qt5Designer.dll
30 | Qt5DesignerComponents.dll
31 | Qt5Gui.dll
32 | Qt5Help.dll
33 | Qt5Multimedia.dll
34 | Qt5MultimediaWidgets.dll
35 | Qt5Network.dll
36 | Qt5OpenGL.dll
37 | Qt5PrintSupport.dll
38 | Qt5Qml.dll
39 | Qt5Quick.dll
40 | Qt5QuickParticles.dll
41 | Qt5QuickTest.dll
42 | Qt5QuickWidgets.dll
43 | Qt5Script.dll
44 | Qt5ScriptTools.dll
45 | Qt5Sql.dll
46 | Qt5Svg.dll
47 | Qt5Test.dll
48 | Qt5Widgets.dll
49 | Qt5Xml.dll
50 | qtjambi5.dll
51 |
52 | +---platforms
53 | |   qminimal.dll
54 | |   qminimald.dll
55 | |   qminimald.pdb
56 | |   qoffscreen.dll
57 | |   qoffscreenend.dll
58 | |   qoffscreenend.pdb
59 | |   qwindows.dll
60 | |   qwindowsd.dll
61 | |   qwindowsd.pdb
62 |
63 | +---plugins
64 | |   +---audio
65 | |   +---bearer
66 | |   +---generic
67 | |   +---iconengines
68 | |   +---imageformats
69 | |   +---mediaservice
70 | |   +---platforms
71 | | |   qminimal.dll
72 | | |   qminimald.dll
73 | | |   qminimald.pdb
74 | | |   qoffscreen.dll
75 | | |   qoffscreenend.dll
76 | | |   qoffscreenend.pdb
77 | | |   qwindows.dll
78 | | |   qwindowsd.dll
79 | | |   qwindowsd.pdb
```

```

80 | |
81 | | +---playlistformats
82 | | +---printsupport
83 | | +---sqldrivers
84 | | \---video
85 | |
86 +---plugins-designer
87 | | +---designer
88 | | | JambiCustomWidget5.dll
89 | | | JambiLanguage5.dll
90 | | \---qtjambi
91 | | | qtjambi_examples.xml
92 | | | qtjambi_widgets.xml|
93 \---qml
94 | +---Qt
95 | +---QtGraphicalEffects
96 | +---QtMultimedia
97 | +---QtQml
98 | +---QtQuick
99 | | +---Controls
100 | | | +---Private
101 | | | \---Styles
102 | | | | qmldir
103 | | | +---Base
104 | | | | \---images
105 | | | +---Desktop
106 | | | \---Flat
107 | | +---Dialogs
108 | | +---Extras
109 | | +---Layouts
110 | | +---LocalStorage
111 | | +---Particles.2
112 | | +---PrivateWidgets
113 | | \---Window.2
114 \---QtQuick.2
115 plugins.qmltypes
116 qmldir
117 qtquick2plugin.dll

```

Listing C.1: Directory structure for a working QtJambi deploy

Appendix D

Final Comparison Table Justifications

	Enforces Clear Architecture	Widgets Availability	Charting Performance	Tooling Quality	Documentation Quality	Can be integrated with Existing Systems	Ease of Development for Clients	Ease of Maintenance for Us	Health Status
JavaFX	✓	✓	✓	✓	✓	✓	✓	✓	-
QtJambi	-	✗	/	✓	✓	-	✓	✗	✗
QtQuick	-	-	✗	✓	✓	-	✓	-	✓
QtWidgets	-	✓	/	✓	✓	-	✗	✗	✓
PyQt5 QtQuick	✓	-	✓	-	✓	-	✓	-	✓
PyQt5 QtWidgets	-	✓	✓	-	✓	-	✓	-	✓

Figure D.1: A qualitative comparison table to sum up the results of the evaluation. The symbols stands for: Tick: Satisfactory or Overly Satisfactory, Dash: Almost Satisfactory, Cross: Non Satisfactory, Slash: Not Assessed.

The parameters listed in the table are:

1. *Patterns Enforced:* Whether the framework helps the developer following some architectural patterns

Final Comparison Table Justifications

2. *Widgets Availability*: Whether the frameworks offers all the widgets our clients need
3. *Charting Performance*: Whether the charting performance is satisfactory
4. *Tooling Quality*: Whether the tooling quality is satisfactory
5. *Documentation Quality*: Whether the documentation quality is satisfactory
6. *Integrates with existing systems*: Whether it integrates with the existing systems and how well
7. *Ease of development for clients*: Whether it involves technologies that our clients know well or are easy to learn
8. *Ease of maintenance*: Whether it involves technologies that are well known in our section or are easy to learn and scale
9. *Health status*: Whether it seems to be growing or at least if it is maintained by the community or some company

D.1 JavaFX

1. *Patterns Enforced*: Yes, mainly MVC
2. *Widgets Availability*: All necessary widgets are available
3. *Charting Performance*: Unsatisfactory in the base version, completely satisfactory with CERN's custom extensions
4. *Tooling Quality*: Overly satisfactory
5. *Documentation Quality*: Satisfactory
6. *Integrates with existing systems*: Completely and smoothly integrates with all our systems
7. *Ease of development for clients*: Clients are supposed to know how to code in Java, therefore it should be suitable.
8. *Ease of maintenance*: Our team is made of Java developers, therefore very suitable.
9. *Health status*: Oracle plans to drop support in a few years and the community around it seems weak.

D.2 QtJambi (for Qt5)

1. *Patterns Enforced*: None
2. *Widgets Availability*: Only very basic widgets could be used
3. *Charting Performance*: Not assessed
4. *Tooling Quality*: Almost satisfactory
5. *Documentation Quality*: Satisfactory
6. *Integrates with existing systems*: It would likely integrate
7. *Ease of development for clients*: Clients are supposed to know how to code in Java, therefore it should be suitable.
8. *Ease of maintenance*: Involves deep knowledge of C++ and JNI, which we do not have: very difficult
9. *Health status*: No community, the project is dead

D.3 QtQuick (in pure JS)

1. *Patterns Enforced*: Yes, MVC
2. *Widgets Availability*: Some complex widgets are lacking functionalities (TreeView, TableView), but are being actively developed
3. *Charting Performance*: Unsatisfactory
4. *Tooling Quality*: Satisfactory
5. *Documentation Quality*: Satisfactory
6. *Integrates with existing systems*: Can weakly integrate through web sockets or C++ custom plugins
7. *Ease of development for clients*: Easy to start with, complex to scale properly due to JavaScript
8. *Ease of maintenance*: Easy to learn, requires us to acquire a lot of new skills for proper maintenance and code management
9. *Health status*: Very actively developed, lively and growing community

D.4 QtWidgets

1. *Patterns Enforced:* None
2. *Widgets Availability:* All necessary widgets are available
3. *Charting Performance:* Not Assessed
4. *Tooling Quality:* Overly satisfactory
5. *Documentation Quality:* Overly Satisfactory
6. *Integrates with existing systems:* Can weakly integrate through web sockets, C++ custom plugins, or JNI
7. *Ease of development for clients:* Very difficult. Clients are not supposed to know how to code in C++ and the language is complex to learn and use properly
8. *Ease of maintenance:* Very difficult. We are not skilled in C++ development.
9. *Health status:* Stable for the foreseeable future.

D.5 PyQt5 QtQuick

1. *Patterns Enforced:* Yes, MVC
2. *Widgets Availability:* Some complex widgets are lacking functionalities (TreeView, TableView), but are being actively developed
3. *Charting Performance:* Satisfactory
4. *Tooling Quality:* Almost Satisfactory, requires some effort to be setup properly
5. *Documentation Quality:* Satisfactory, relies completely on the C++ one
6. *Integrates with existing systems:* Can integrate partially through Python interfaces, support for Python from the backend is steadily increasing
7. *Ease of development for clients:* Clients are not supposed to know Python but most of them do. In addition, is easy to learn.
8. *Ease of maintenance:* We are not supposed to know Python, but we can work with it and learn fast.

9. *Health status*: Quite healthy, gaining momentum along with pure QtQuick

D.6 PyQt5 QtWidgets

1. *Patterns Enforced*: None
2. *Widgets Availability*: All necessary widgets are available
3. *Charting Performance*: Satisfactory
4. *Tooling Quality*: Almost Satisfactory, requires some effort to be setup properly
5. *Documentation Quality*: Satisfactory, relies completely on the C++ one
6. *Integrates with existing systems*: Can integrate partially through Python interfaces, support for Python from the backend is steadily increasing
7. *Ease of development for clients*: Clients are not supposed to know Python but most of them do. In addition, is easy to learn.
8. *Ease of maintenance*: We are not supposed to know Python, but we can work with it and learn fast.
9. *Health status*: Quite healthy, stable and supported for the foreseeable future.

Bibliography

- [1] Chua Hock-Chuan, *Java Programming Tutorial: Programming Graphical User Interface (GUI)*, https://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html at yet another insignificant... programming notes (<https://www3.ntu.edu.sg/home/ehchua/programming/index.html>)
- [2] Christiane Lefèvre, *The CERN accelerator complex*, <https://cds.cern.ch/record/1260465> at CERN Document Server (<https://cds.cern.ch>)
- [3] *Java Client Roadmap Update (page 6)* <http://www.oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf> at Oracle (<https://www.oracle.com>)
- [4] *StackOverflow Trends*, <https://insights.stackoverflow.com/trends?tags=qt%2Cjavafx%2Cswing%2Cpyqt%2Cgtk%2Cangular%2Creactjs> at StackOverflow Insights (<https://insights.stackoverflow.com>)
- [5] *Google Trends Search*, <https://trends.google.com/trends/explore?date=today%205-y&q=qt,javafx,swing,gtk,pyqt> at Google Trends (<https://trends.google.com>)
- [6] *OpenJFX Home Page* <https://wiki.openjdk.java.net/display/OpenJFX/Main> at OpenJDK (<https://wiki.openjdk.java.net>)
- [7] *Qt Home Page* <https://www.qt.io/>
- [8] *Qt Customer Success Stories* <https://resources.qt.io/customer-stories-all>
- [9] *KDE Home Page* <https://www.kde.org/>

BIBLIOGRAPHY

- [10] *About: The Qt Company* <https://www.qt.io/company>
- [11] *Rapid Application Development* https://en.wikipedia.org/wiki/Rapid_application_development at Wikipedia (<https://en.wikipedia.org>)
- [12] A. Powell, A. Vickers, *A practical strategy for the evaluation of software tools* in S. Brinkkemper et al. (eds.), *Method Engineering*, Springer Science+Business Media Dordrecht, 1996 (retrieved at https://link.springer.com/content/pdf/10.1007/978-0-387-35080-6_11.pdf)
- [13] I. Allen, *The Brutal Lifecycle of JavaScript Frameworks* <https://stackoverflow.blog/2018/01/11/brutal-lifecycle-javascript-frameworks/> at StackOverflow Blog (<https://stackoverflow.blog>)
- [14] *What do we do?* <https://software.ac.uk/what-do-we-do> at Software Sustainability Institute (<https://software.ac.uk/>)
- [15] *Software Evaluation Guide* <https://www.software.ac.uk/resources/guides-everything/software-evaluation-guide> at Software Sustainability Institute (<https://software.ac.uk/>)
- [16] *How To Evaluate Software* <https://bi-survey.com/software-evaluation> at BI-Survey.com (<https://bi-survey.com/>)
- [17] *Software Evaluation Criteria Template - Google Search* <https://www.google.com/search?q=software+evaluation+criteria+template>
- [18] *e(fx)clipse: JavaFX Tooling and Runtime for Eclipse and OSGi* <http://www.eclipse.org/efxclipse/index.html> at Eclipse (<http://www.eclipse.org/>)
- [19] *Download Scene Builder* <https://gluonhq.com/products/scene-builder/#download> at Gluon (<https://gluonhq.com/>)
- [20] G. Kruk, M. Peryt, *JDATAVIEWER – JAVA BASED CHARTING LIBRARY* <http://cds.cern.ch/record/1215878/files/CERN-ATS-2009-111.pdf?version=1> at CERN Document Server (<http://cds.cern.ch/record/1215878/>)

-
- [21] *JDVE - Demos* <https://wikis.cern.ch/display/InCA/JDVE++Demos> at CERN Wikis (<https://wikis.cern.ch/>)
- [22] *Java Platform, Standard Edition (Java SE) 8: Client Technologies* <https://docs.oracle.com/javase/8/javase-clienttechnologies.htm> at Java Platform, Standard Edition (Java SE) 8 Documentation (<https://docs.oracle.com/javase/8/>)
- [23] *Getting Started with JavaFX* <https://docs.oracle.com/javase/8/javafx/JFXST.pdf> at Java Platform, Standard Edition (Java SE) 8 Documentation (<https://docs.oracle.com/javase/8/>)
- [24] *Model-view-controller* <https://en.wikipedia.org/wiki/Model-view-controller> at Wikipedia (<https://en.wikipedia.org>)
- [25] *Understanding the JavaFX Architecture* <https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm#A1106498> at Java Platform, Standard Edition (Java SE) 8 Documentation (<https://docs.oracle.com/javase/8/>)
- [26] *Working with the JavaFX Scene Graph* <https://docs.oracle.com/javase/8/javafx/scene-graph-tutorial/scenegraph.htm#JFXSG107> at Java Platform, Standard Edition (Java SE) 8 Documentation (<https://docs.oracle.com/javase/8/>)
- [27] *Implementing JavaFX Best Practices* https://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.htm at Java Platform, Standard Edition (Java SE) 8 Documentation (<https://docs.oracle.com/javase/8/>)
- [28] *Scene Builder* <http://gluonhq.com/products/scene-builder/> at Gluon (<http://gluonhq.com/>)
- [29] G.Kruk, O.Alves, L.Molinari *JavaFX Charts: Implementation of missing features* <http://cds.cern.ch/record/2305669/files/tupha186.pdf> at CERN Document Server (<http://cds.cern.ch/>)
- [30] *Qt (software)* [https://en.wikipedia.org/wiki/Qt_\(software\)](https://en.wikipedia.org/wiki/Qt_(software)) at Wikipedia (<https://en.wikipedia.org>)

BIBLIOGRAPHY

- [31] *Qt Licensing* <http://doc.qt.io/qt-5/licensing.html> at Qt5 Documentation (<https://doc.qt.io/qt-5/>)
- [32] *QT GROUP OYJ - Managers' Transactions, 12/4/2017* <https://investors.qt.io/investor-services/releases/?release=A84F7B0247145FCF> at Qt for Investors (<https://investors.qt.io/>)
- [33] *qmlRegisterType (Method Documentation)* <https://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType> at Qt5 Documentation (<https://doc.qt.io/qt-5/>)
- [34] *setContextProperty (Method Documentation)* <https://doc.qt.io/qt-5/qqmlcontext.html#setContextProperty> at Qt5.10 Documentation (<https://doc.qt.io/qt-5/>)
- [35] *Using the Meta-Object Compiler (moc)* <https://doc.qt.io/qt-5/moc.html> at Qt5 Documentation (<https://doc.qt.io/qt-5/>)
- [36] *The Qt Company* <https://www.qt.io/company>
- [37] *Qt Quarterly 29* <https://doc.qt.io/archives/qq/QtQuarterly29.pdf> at Qt Quarterly Archives (<https://doc.qt.io/archives/qq/>)
- [38] *Signals & Slots* <https://doc.qt.io/qt-5/signalsandslots.html> at Qt5.10 Documentation (<https://doc.qt.io/qt-5/>)
- [39] *Qt macros: Q_PROPERTY* https://doc.qt.io/qt-5/qobject.html#Q_PROPERTY at Qt5.10 Documentation (<https://doc.qt.io/qt-5/>)
- [40] *User Interface Compiler (uic)* <https://doc.qt.io/qt-5/uic.html> at Qt5.10 Documentation (<https://doc.qt.io/qt-5/>)
- [41] *The Meta-Object System* <https://doc.qt.io/qt-5/metaobjects.html> at Qt5.10 Documentation (<https://doc.qt.io/qt-5/>)
- [42] Jacques Malenfant et al. *A Tutorial on Behavioral Reflection and its Implementation* <http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf>
- [43] *Qt Downloads* <https://www.qt.io/download>

- [44] *Offline Qt Downloads* <https://www1.qt.io/offline-installers/>
- [45] *Qt Creator Manual* <http://doc.qt.io/qtcreator/>
- [46] *Profiling QML Applications* <https://doc.qt.io/qtcreator/creator-qml-performance-monitor.html> at QtCreator Documentation (<https://doc.qt.io/qtcreator/>)
- [47] *Qt Wiki* <https://wiki.qt.io/Main>
- [48] *Use TypeScript to write GUI logic in Qt Quick (instead of JavaScript or C++)* <https://bugreports.qt.io/browse/QTBUG-68810> at Qt Bug Tracker (<https://bugreports.qt.io>)
- [49] *Qt for Python* <https://www.qt.io/qt-for-python>
- [50] *Anaconda Package List* https://docs.anaconda.com/anaconda/packages/py3.7_linux-64/
- [51] *PyQt5* <https://pypi.org/project/PyQt5/> at PyPI (<https://pypi.org>)
- [52] *Installing PyQt5* <https://pyqt.readthedocs.io/en/latest/installation.html#building-and-installing-from-source> at PyQt v5.11.2 Reference Guide (<https://pyqt.readthedocs.io>)
- [53] *What is PyQt?* <https://www.riverbankcomputing.com/software/pyqt/intro> at Riverbank (<https://www.riverbankcomputing.com/>)
- [54] *Qt Jambi: Qt for Java* <http://old.qt-jambi.org/>
- [55] *Qt Jambi Reference Documentation* https://doc.qt.io/archives/qtjambi-4.5.2_01/
- [56] *QtJambi5* <https://github.com/OmixVisualization/qtjambi5> at GitHub (<https://github.com/>)
- [57] *QtJambi5* <https://github.com/tilialabs/qtjambi5> at GitHub (<https://github.com/>)
- [58] *Java User Interface Compiler (JUIC)* https://doc.qt.io/archives/qtjambi-4.5.2_01/com/trolltech/qt/qtjambi-juic.html at Qt4.5 Documentation (https://doc.qt.io/archives/qtjambi-4.5.2_01)

BIBLIOGRAPHY

- [59] *The Qt Jambi Generator* https://doc.qt.io/archives/qtjambi-4.5.2_01/com/trolltech/qt/qtjambi-generator.html at Qt Documentation Archives (https://doc.qt.io/archives/qtjambi-4.5.2_01/)
- [60] *Tilialabs Home Page* <http://tilialabs.com/>
- [61] *QtJambi5 (kkofler's repo)* <https://github.com/kkofler/qtjambi5/tree/dagopt> at GitHub (<https://github.com/>)
- [62] *How to use Qt Jambi generator* <http://old.qt-jambi.org/doc/generator> at QtJambi Homepage (<http://old.qt-jambi.org>)
- [63] *How to use Qt Jambi generator* https://doc.qt.io/archives/qtjambi-4.5.2_01/com/trolltech/qt/qtjambi-generator.html at the Qt Documentation (<https://doc.qt.io/>)
- [64] *Qt Jambi Generator Example* https://doc.qt.io/archives/qtjambi-4.5.2_01/com/trolltech/qt/qtjambi-generatorexample.html at the Qt Documentation (<https://doc.qt.io/>)
- [65] *Qt Jambi Eclipse Integration* https://doc.qt.io/archives/qtjambi-4.5.2_01/com/trolltech/qt/qtjambi-eclipse.html at Qt Documentation Archives (https://doc.qt.io/archives/qtjambi-4.5.2_01/)
- [66] *PyDM - Installation* <https://slaclab.github.io/pydm/installation.html#troubleshooting-pydm-widgets-in-designer>
- [67] *Riverbank Homepage* <https://www.riverbankcomputing.com/news>
- [68] *Matplotlib Home Page* <https://matplotlib.org/>
- [69] *Seaborn Home Page* <https://seaborn.pydata.org/>
- [70] *Matplotlib QtQuick Playground* <https://github.com/fcollonval/matplotlib-qtquick-playground>
- [71] Kevin Shing Bruce Li, *PySPSDamperApp* <https://gitlab.cern.ch/kli/PySPSDamperApp>
- [72] *Python PIP* https://www.w3schools.com/python/python_pip.asp at W3Schools.com (<https://www.w3schools.com>)

- [73] *pip 18.0 Documentation* <https://pip.pypa.io/en/stable/> at PyPA (<https://www.pypa.io/en/latest/>)
- [74] Sara Zanzottera, *QML + Matplotlib integration for PyQt* <https://gitlab.cern.ch/acc-co/accsoft/gui/qt-evaluation/qt-pyqt-integration/tree/master/matplotlib-qml-backend>
- [75] *PyCharm Home Page* <https://www.jetbrains.com/pycharm/>
- [76] *Visual Studio Code Home Page* <https://code.visualstudio.com/>
- [77] *Gedit* <https://wiki.gnome.org/Apps/Gedit> at the GNOME Wiki (<https://wiki.gnome.org/>)
- [78] *Python Extension for VSCode* <https://marketplace.visualstudio.com/items?itemName=ms-python.python> at the Visual Studio Marketplace (<https://marketplace.visualstudio.com>)
- [79] *QML Extension for VSCode* <https://marketplace.visualstudio.com/items?itemName=bbenoist.QML> at the Visual Studio Marketplace (<https://marketplace.visualstudio.com>)
- [80] *Qt for Python 5.11 released* <https://blog.qt.io/blog/2018/06/13/qt-python-5-11-released/> at Qt Blog (<https://blog.qt.io/>)
- [81] *Evolution of the QML engine, part 1* <https://blog.qt.io/blog/2013/04/15/evolution-of-the-qml-engine-part-1/> at Qt Blog (<https://blog.qt.io/>)
- [82] *Importing JavaScript Resources in QML* <https://doc.qt.io/qt-5/qtqml-javascript-imports.html> at the Qt Documentation (<https://doc.qt.io/>)
- [83] *JavaScript Host Environment* <https://doc.qt.io/qt-5/qtqml-javascript-hostenvironment.html> at the Qt Documentation (<https://doc.qt.io/>)
- [84] *QML Global Object* <https://doc.qt.io/qt-5/qtqml-javascript-qmlglobalobject.html> at the Qt Documentation (<https://doc.qt.io/>)

BIBLIOGRAPHY

- [85] *List of JavaScript Objects and Functions* <https://doc.qt.io/qt-5/qtqml-javascript-functionlist.html> at the Qt Documentation (<https://doc.qt.io/>)
- [86] *JavaScript Environment Restrictions* <https://doc.qt.io/qt-5/qtqml-javascript-hostenvironment.html#javascript-environment-restrictions> at the Qt Documentation (<https://doc.qt.io/>)
- [87] *Standard ECMA-262* <http://www.ecma-international.org/publications/standards/Ecma-262.htm> at ECMA International (<http://www.ecma-international.org/>)
- [88] *WorkerScript QML Type* <http://doc.qt.io/qt-5/qml-workerscript.html> at the Qt Documentation (<https://doc.qt.io/>)
- [89] *Compiler Options* <https://www.typescriptlang.org/docs/handbook/compiler-options.html> at TypeScriptLang.org (<https://www.typescriptlang.org>)
- [90] *About Node.js* <https://nodejs.org/en/about/> at Node.js (<https://nodejs.org/en/>)
- [91] *NPM Homepage* <https://www.npmjs.com/>
- [92] *Brig's GitHub Repository* <https://github.com/BrigJS/brig> at GitHub (<https://github.com>)
- [93] *Quickly's GitHub Repository* <https://github.com/quickly/quickly> at GitHub (<https://github.com>)
- [94] *Qt Charts 2.1.0 Release* <http://blog.qt.io/blog/2016/01/18/qt-charts-2-1-0-release/> at Qt Blog (<http://blog.qt.io/>)
- [95] *Qt Charts Overview* <https://doc.qt.io/qt-5.11/qtcharts-overview.html> at the Qt Documentation (<https://doc.qt.io/>)
- [96] *Qwt User's Guide* <http://qwt.sourceforge.net/>
- [97] *QCustomPlot Homepage* <https://www.qcustomplot.com/>

BIBLIOGRAPHY

- [98] *How to scroll a QChart as realtime data come in?* <http://www.qtcentre.org/threads/69063-How-to-scroll-a-QChart-as-realtime-data-come-in> at Qt Centre (<http://www.qtcentre.org>)
- [99] *LineSeries QML Type* <https://doc.qt.io/qt-5/qml-qtcharts-lineseries.html> at Qt Documentation (<https://doc.qt.io/>)
- [100] *AreaSeries QML Type* <https://doc.qt.io/qt-5/qml-qtcharts-areaseries.html> at Qt Documentation (<https://doc.qt.io/>)
- [101] *QCharts Second Y Axis on the Right Side* <https://stackoverflow.com/questions/50956222/qcharts-second-y-axis-on-the-right-side> at Stack Overflow (<https://stackoverflow.com>)
- [102] *Append negative CategoryAxis Labels* <https://forum.qt.io/topic/93688/append-negative-categoryaxis-labels> at Qt Forum (<https://forum.qt.io>)
- [103] *QML Integration* <https://www.qcustomplot.com/index.php/support/forum/172> at QCustomPlot Discussion and Comments (<https://www.qcustomplot.com/index.php/support/forum>)
- [104] *Emanuel Eichhammer GitLab Profile* <https://gitlab.com/DerManu> at GitLab (<https://gitlab.com/>)
- [105] *ElectronJS Homepage* <https://electronjs.org/>
- [106] *The Abstract Windowing Toolkit Group*, <http://openjdk.java.net/groups/awt/> at OpenJDK (<http://openjdk.java.net/>)
- [107] *java.awt Package Documentation*, <https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html> at Java™ Platform, Standard Edition 7 API Specification (<https://docs.oracle.com/javase/7/docs/api>)

BIBLIOGRAPHY

- [108] *The Swing Tutorial*, <https://docs.oracle.com/javase/tutorial/uiswing/> at The Java™ Tutorials (<https://docs.oracle.com/javase/tutorial/>)
- [109] *javax.swing Package Documentation*, <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html> at Java™ Platform, Standard Edition 7 API Specification (<https://docs.oracle.com/javase/7/docs/api>)
- [110] *General information on JavaFX*, <https://www.java.com/en/download/faq/javafx.xml> at Java.com (<https://www.java.com>)
- [111] *SWT: The Standard Widget Toolkit*, <https://www.eclipse.org/swt/> at Eclipse.org (<https://www.eclipse.org/>)
- [112] *GWT (Google Widget Toolkit)*, <http://www.gwtproject.org/>
- [113] *JOGL (Java OpenGL Binding)*, <http://jogamp.org/jogl/www/> at JogAmp.org (<http://jogamp.org>)
- [114] *Continuous Integration*, https://en.wikipedia.org/wiki/Continuous_integration at Wikipedia (<https://en.wikipedia.org>)
- [115] *What is Continuous Integration?*, <https://www.visualstudio.com/fr/learn/what-is-continuous-integration/> at VisualStudio.com (<https://www.visualstudio.com/>)
- [116] *WYSIWYG*, <https://en.wikipedia.org/wiki/WYSIWYG> at Wikipedia (<https://en.wikipedia.org/>)
- [117] *Integrated Development Environment*, https://en.wikipedia.org/wiki/Integrated_development_environment at Wikipedia (<https://en.wikipedia.org/>)
- [118] *SWIG Homepage*, <http://swig.org/>
- [119] *Hello World, JavaFX Style* https://docs.oracle.com/javafx/2/get_started/hello_world.htm at Java Platform, Standard Edition (Java SE) 8 Documentation (<https://docs.oracle.com/javase/8/>)

BIBLIOGRAPHY

- [120] *Creating a minimal JavaFX user interface in Java 8* <https://gjf2a.blogspot.fr/2015/01/creating-minimal-javafx-user-interface.html> at Computing Intelligently (<https://gjf2a.blogspot.fr/>)
- [121] *Software development kit* https://en.wikipedia.org/wiki/Software_development_kit at Wikipedia (<https://en.wikipedia.org>)