Institute for Computer Graphics and Vision
Graz University of Technology, Austria

Master's Thesis

A Modern Graphical User Interface Framework for Air Traffic Control Information Systems

Bernhard Roth, BSc

January 18, 2013

Supervised by
Dipl.-Ing. Dr.techn. Bernhard Kainz
and
Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Abstract

Air traffic controllers are used to observe a vast amount of different systems with inconsistent user interfaces. In this thesis we present the design of a client server architecture to integrate these systems into one that provides a homogeneous graphical user interface. The primary goals of the framework are adaptation flexibility, rapid prototyping capabilities, to be able to involve controllers in early project phases and the simple application of user interface design principles to optimize situational awareness. Amongst others, these principles, which we summarize in this work, include the use of colors, animations and shapes.

Instead of using conventional toolkits for desktop application development, the graphical user interface of the presented system is built upon QtQuick, a library to create arbitrary user experiences through a declarative language, without the need for constant compilation. In this work we discuss details on the technology's advantages and disadvantages and give reasons for our motivation to use it.

We explain the system's design, paired with additional implementation details and present several prototypes, created with it, to demonstrate its possibilities. These prototypes are evaluated in regard to project adaptation efforts and usability impressions of controllers from different sites in the world, where the presented system will be installed in the near future.

The presented framework delivers low adaptation times and flexible capabilities to apply user interface design metaphors, which makes it well suitable for the intended use. In this regard, QtQuick proved to be a solid basis for the system.

Keywords: Air Traffic Control, ATC Information System, Graphical User Interface, User Interface Design, Human Computer Interaction, Prototyping, UI Toolkits, QtQuick, QML, Automation, Systems Integration, Situational Awareness

Zusammenfasssung

Fluglotsen sind es gewohnt, eine Vielzahl verschiedener Systeme mit inkonsistenten Benutzeroberflächen im Auge zu behalten. In dieser Arbeit präsentieren wir das Design einer Client-Server Architektur, um diese Systeme in ein einzelnes, mit homogener Benutzeroberfläche, zu integrieren. Die primären Ziele des Frameworks sind flexible Anpassungsmöglichkeiten, die Fähigkeit, schnell Prototypen erstellen zu können, um Lotsen schon in frühen Projektphasen einbinden zu können und die leichte Anwendbarkeit von Design-Prinzipien für Benutzeroberflächen. Unter anderem beinhalten diese Prinzipien, die wir in dieser Arbeit zusammenfassen, die Nutzung von Farben, Animationen und Formen.

Anstatt eines der konventionellen Toolkits für die Desktop-Anwendungsentwicklung zu verwenden, baut die graphische Benutzeroberfläche des präsentierten Systems auf QtQuick auf. Dies ist eine Bibliothek um beliebige Oberflächen mit Hilfe einer deklarativen Sprache zu erstellen, ohne den Programmcode ständig neu übersetzen zu müssen. In dieser Arbeit erläutern wir unsere Motivation, diese Technologie zu nutzen und besprechen Details über ihre Vor- und Nachteile.

Wir erklären das Design und zusätzliche Details der Implementierung des Systems und präsentieren mehrere Prototypen, die damit erstellt wurden, um seine Möglichkeiten zu zeigen. Wir evaluieren diese Prototypen in Bezug auf die Aufwände, die es benötigt um sie an Projekte anzupassen. Ebenfalls evaluieren wir Eindrücke über die Benutzerfreundlichkeit des Systems. Diese Eindrücke stammen von Fluglotsen von mehreren Flughäfen dieser Welt, wo das System in näherer Zukunft installiert wird.

Das vorgestellte Framework ermöglicht niedrige Anpassungszeiten und flexible Möglichkeiten die Prinzipien des Benutzeroberflächendesigns anzuwenden. Dieses Ergebnis zeigt, dass sich das System für den gedachten Einsatzzweck gut eignet und dass QtQuick sich als Basis für ein solches Framwork bewährt hat.

Schlüsselwörter: Flugsicherung, ATC Informationssysteme, Graphische Benutzeroberflächen, Benutzeroberflächendesign, Mensch-Maschine Interaktion, Prototyping, UI Toolkits, QtQuick, QML, Automatisierung, Systemintegration, Situationsbewusstsein

Contents

Figures and Tables **xiii**
List of Figures . . . xiii
List of Tables . . . xiv

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Structure . . . 2
1.3 Air Traffic Control . . . 3
1.3.1 Who is responsible? . . . 4
1.3.2 The Different Areas of Responsibility . . . 4
1.3.3 The Departure . . . 5
1.3.4 The Flight . . . 7
1.3.5 The Arrival . . . 7
1.4 Technical Background of Air Traffic Control . . . 8
1.4.1 Voice Communication Systems . . . 8
1.4.2 ATIS . . . 9
1.4.3 Positioning Systems . . . 9
1.4.4 Weather Systems . . . 10
1.4.5 Airfield Lighting Systems . . . 11
1.4.6 ILS & Navigational Aids . . . 11
1.4.7 Electronic Flight Strips . . . 11
1.4.8 Summary . . . 12

2 Related Work **13**
2.1 Difficulties of ATC Automation . . . 13
2.2 ATM System Developer's Survival Strategies . . . 15
2.2.1 Bringing the Controllers on Board . . . 15
2.2.2 Providing the Best Possible User Experience . . . 16
2.3 User Interface Design in the Context of ATC . . . 17
2.3.1 Colors . . . 17
2.3.2 Animation . . . 18
2.3.3 Fonts . . . 19
2.3.4 Shapes . . . 20
2.3.5 Sounds . . . 20
2.4 ATM System Development . . . 20
2.5 Existing Tower Information Systems . . . 21

3 Methodolgy 25
3.1 Requirements Specification . . . 25
3.1.1 Requirements and Goals . . . 25
3.2 Technology Evaluation and Decision . . . 31
3.3 System Design . . . 31
3.4 Evaluation Method . . . 31

4 System Overview, Design and Implementation 33
4.1 Technology Overview and Evaluation . . . 33
4.1.1 The Qt Library . . . 33
4.1.2 The QtQuick Module . . . 34
4.1.3 QtQuick Evaluation Details . . . 34
4.1.3.1 Custom Extensions to QtQuick . . . 35
4.1.3.2 QtQuick HMI with C++ Back-End . . . 37
4.1.3.3 Custom HMI Component Management . . . 38
4.1.3.4 Editor Capabilities . . . 38
4.1.3.5 Evaluation Summary . . . 39
4.2 System Design . . . 39
4.2.1 Design Overview . . . 39
4.2.2 Communication Design . . . 41
4.2.3 Data Item Design . . . 43
4.2.4 Client Design . . . 44
4.2.5 Server Design . . . 47
4.2.6 Interface Process Design . . . 49
4.3 Implementation Details . . . 50
4.3.1 Overall Details . . . 50
4.3.2 Client Details . . . 51
4.3.3 Server Details . . . 55
4.3.4 Interface Process & Simulator Details . . . 58

5 Results 61
5.1 Evaluation Scheme . . . 61
5.2 Efforts Analysis Preface . . . 62
5.3 Core Framework . . . 64
5.4 Prototype Results . . . 64
5.4.1 System Type 1 - The Typical System . . . 65
5.4.1.1 Version 1 - Saudi Arabia . . . 66
5.4.1.2 Version 2 - Parchim Airport, Germany . . . 71
5.4.2 System Type 2 - The Special Purpose System . . . 75
5.5 Efforts Summary . . . 82
5.6 Discussion . . . 83

6 Conclusion and Future Work 85
6.1 Conclusion . . . 85

6.2 Future Work 88

Acronyms **91**

Bibliography **93**

Figures and Tables

List of Figures

1.1 Traditional Controller Working Positions . . . 2
1.2 ATC Responsibilities . . . 5
1.3 Comparison of Paper Flight Strips and Electronic Flight Strips . . . 11

2.1 Market Comparison: Nav Canada . . . 22
2.2 Market Comparison: ACAMS AS . . . 23
2.3 Market Comparison: Frquentis . . . 23

4.1 Existing Tower Information System . . . 36
4.2 Design Overview: System . . . 40
4.3 Examples of the Data Item's Dot Notation . . . 44
4.4 Design Overview: Client . . . 45
4.5 Design Overview: Server . . . 47
4.6 Design Overview: Interface & Simulator Processes . . . 50
4.7 Class Diagram: Client . . . 52
4.8 Class Diagram: Server . . . 55
4.9 Class Diagram: Interface & Simulator Processes . . . 58

5.1 Saudi Arabian Prototype: Main Page . . . 66
5.2 Saudi Arabian Prototype: Pages Besides the Main Page . . . 67
5.3 Saudi Arabian Prototype: Different Main Pages of Saudi Arabian Sites . . . 69
5.4 Color Scheme Examples . . . 70
5.5 Saudi Arabian Prototype: Windshear Pop-Up . . . 70
5.6 Parchim Airport Prototype: Main Page . . . 73
5.7 Parchim Airport Prototype: AFL Bar Integration into Other System . . . 74
5.8 Parchim Airport Prototype: Special Invisible Button Behaviour . . . 75
5.9 Parchim Airport Prototype: Concepts of NOTAM Pages . . . 76
5.10 Austrian Prototype: Individual Local Pages . . . 77
5.11 Austrian Prototype: Special Shared Pages . . . 78
5.12 Austrian Prototype: Field Change Interaction Sequence . . . 79
5.13 Austrian Prototype: Indication of Possible Interaction . . . 80
5.14 Austrian Prototype: Frequency Editing Workflow Sequence . . . 81

List of Tables

5.1 Overall Efforts Overview 63
5.2 Core Framework Efforts 64
5.3 Efforts Overview for Prototypes 65
5.4 Project Adaptation Efforts with Older Framework 83

Chapter 1

Introduction

Since mankind has found a way to travel by plane, the skies have seen an ever increasing amount of traffic. This ongoing boost results in the demand of more and more efforts to ensure that safety is not impaired by the rising number of flights. Since the capacities of human air traffic controllers are limited, computer based solutions have gained a large importance in assisting in the decision making process of air traffic controllers.

AviBit Data Processing GmbH, (simply called AviBit for the remainder of this work) a software company situated in Graz, Austria that develops Air Traffic Control (ATC) solutions, has asked the Institute for Computer Graphics and Vision at Graz University of Technology, for support in the development and evaluation of a framework for a new Tower Information System (TIS). The results of this research are the topic of this master's thesis.

1.1 Motivation

AviBit is a small to medium sized company that provides Air Navigation Service Providers (ANSPs) with different systems, used to reduce the workload of air traffic controllers.

In general most of the systems used in ATC are fitted to a specific purpose, like showing the current traffic situation on the ground, or managing flight plans. In contrast to these special purpose applications, the proposed result of this work is a framework for a more general supporting system. Such a TIS should act as an interface to as many supporting systems, available at an airport, as possible and combine them all in one consistent user interface experience, to integrate most of the information and controls, currently distributed among many different systems on the controller's working position. Usually there is a multitude of different information available at any given site, which tends to clutter the controller's working environment with a lot of different controls and displays, as can be seen in Figure 1.1. The goal of this project is therefore, to develop a solution, which makes the integration of many different data sources (i.e., interfaces to data providers) and controls as easy and extensible as possible. This shall lower the time spent for AviBit, to adapt the product to a given site. Any site can have a large range of different systems in place, which all provide a special purpose user interface. All of these systems are typically delivered by different vendors and the process of integrating them is not only impeded by technical problems, which pose real engineering efforts, like the processing of radar data, but oftentimes it is due to

Figure 1.1: *These images show the traditional look of an ATCO's working position. We see that the whole working environment is cluttered with a multitude of different systems that seem to be placed wherever some room was left.*

commercial reasons that system integration gets tedious. Most system-vendors are not very pleased with the idea of another company wanting to cut off some of their cake, so the problem of missing connection possibilities to external systems is a serious one. Even if there are interfaces available, they often use undocumented proprietary protocols, which pose more problems, than the oftentimes relatively simple implementation of a known protocol. Anyway, even if the commercial and strategic considerations involved in ATC projects are a big aspect of the topic of integrating all those systems, this aspect does not gain any focus in this thesis. The only commercial part relevant to this project is discussed in Chapter 5 (Results) of this thesis. There we analyze the impact of the developed framework on the adaptation times to a new site, compared to a previously developed product from AviBit.

The main focus of this work lies on the graphical user interface of such a solution and its implementation but since the concept would only be half-complete without mentioning the tightly coupled data-side of the framework, we will also discuss this part in Chapter 4 (System Overview, Design and Implementation).

In this work we present a solution to make integration of different systems, as well as Human-Machine Interface (HMI) customization, as simple as possible while maintaining large flexibility, in fulfilling the user's needs and expectations from a tailor-made solution.

1.2 Structure

The remainder of this first chapter addresses the topic of air traffic control in general to create a deeper understanding of the workflows and complexity this industry has to deal with. It explains the different parties involved in handling the safe movement of aircraft from one airport to another. In this chapter we also introduce different ATC related systems available, describe their use cases and give a more in-depth overview on the specific problems they try to solve. With this introduction we explain the problem

domain and which part of it we cover with this work.

Chapter 2 (Related Work) deals with the difficulties software engineers have to face when confronting Air Traffic Controllers (ATCOs) with new systems and presents proposals to solve these problems. It also deals with the important concept of situational awareness and principles of user interface design that can be applied to achieve it. The focus lies on the special needs that result from the ATC environment where the framework, presented in this work, shall be used.

Subsequently, Chapter 3 (Methodolgy) explains the methods used to approach this project. Since this work is based on AviBit's demand to create a new product they had in their mind, this chapter also deals with the many different requirements the company has for such a system. Many of these requirements are based on experiences of the past and others are the result of considerations concerning the current and future development of the ATC business as well as implications that arise from the technological trends of the last years.

After an extensive description of the requirements in the previous chapter, Chapter 4 (System Overview, Design and Implementation) covers our ideas on how to realize the HMI part and the other modules of the framework, according to the requirements captured before. We address the software creation process itself, from the prototyping and technology evaluation phase, to the design and the final implementation. Therefore, we cover details of the chosen technology, and explain the design of the system as well as certain implementation details.

In Chapter 5 (Results) we discuss the evaluation scheme we prepared for the resulting framework and the results themselves, which focus on the outcomes of three prototypes, created with the presented framework.

As the final part of this work, Chapter 6 (Conclusion and Future Work) summarizes the conclusions drawn from this work and gives an outlook on possible improvements of the system.

1.3 Air Traffic Control

To better understand the remainder of this thesis this section provides an overview about air traffic control. There is a lot of domain specific knowledge necessary to fully comprehend the workflows involved in the process of guiding an aircraft safely from the airport of departure to its airport of destination. It is impossible to cover all these topics in a publication like this but we address all details necessary to understand the goals of this work; even if the reader does not have any previous knowledge of the air traffic domain. This has to be done, as a better understanding of the processes related to the goals of ATC, will provide a deeper knowledge for the audience of this thesis, which will in turn help us to outline the intentions of the framework, realized during this project. As already mentioned it is not possible to explain every detail and for this reason we focus on the workflows more closely related to the airport, than on the procedures that are important for pilots. The latter and other related topics are only mentioned by the way during this chapter since it would otherwise go far beyond the scope of this thesis.

Another important thing to mention is that we solely treat civil aviation and do not cover military specifics, which might differ significantly in terms of procedures.

1.3.1 Who is responsible?

People may think that the company which is responsible for operating the airport is also responsible for the safe handling of the planes starting and landing there. In fact this is not the case. This process is instead handled by ANSPs. Known ANSPs are, for example, AustroControl GmbH (ACG), which handles austrian airports or the Deutsche Flugsicherung GmbH (DFS), which is responsible for airports in Germany. These companies are not necessarily only operating in exact one country, as ACG, for example, is also responsible for some regional airports in Germany; in respect to german laws and regulations. ANSPs can be controlled by governmental organizations as well as private or semi-private companies. There are many connection points between the airport operator or authority and the ANSP, which have to be coordinated in detail, to provide an efficient workflow and avoid any delays or other inconveniences for the passengers, which should in the best case not even notice the difficulties and complexity of optimizing air traffic around the globe. These connection points are, for example, related to fueling and cleaning aircraft in time, apron management, which means the area where the parking positions for aircraft are located, passenger boarding, which also has to happen in time and other tasks that are carried out by the airport's operator. But this work focuses on air traffic control and not on the difficulties of airport logistics, therefore, we assume in our case that all passengers are boarded, the aircraft is fueled accordingly and everyone on board is waiting to get the journey started.

The following subsections describe the way of an aircraft from its departure gate to its arrival gate as comprehensive as possible, without delving into too many details that are not relevant to understand this work. In the same time the different types of controllers are introduced and their duties are explained. The explanations provided, do in reality not match exactly for every airport or airspace but since we do not explain the exact procedures, which can differ from country to country, they are sufficient to get a profound knowledge of the chain of responsibility in the air and the austrian airspace especially.

Most of the procedures used in ATC all over the world are based on standards, rules and guidelines created by the International Civil Aviation Organization (ICAO), which is a worldwide organization that is, amongst other fields of activity, responsible for improvements in air traffic safety as well as air space management.

1.3.2 The Different Areas of Responsibility

The responsibilities in controlling aircraft to guarantee safe and collision free movements during the whole travel are split up between different parties involved. Depending on the size of the airport in question or the amount of traffic in a specific airspace these parties can be separate entities or combined to be handled by the same people, or centers respectively. Nevertheless, their roles and responsibilities are well defined as described in the following sections. In addition Figure 1.2 shows a diagram of the responsible units

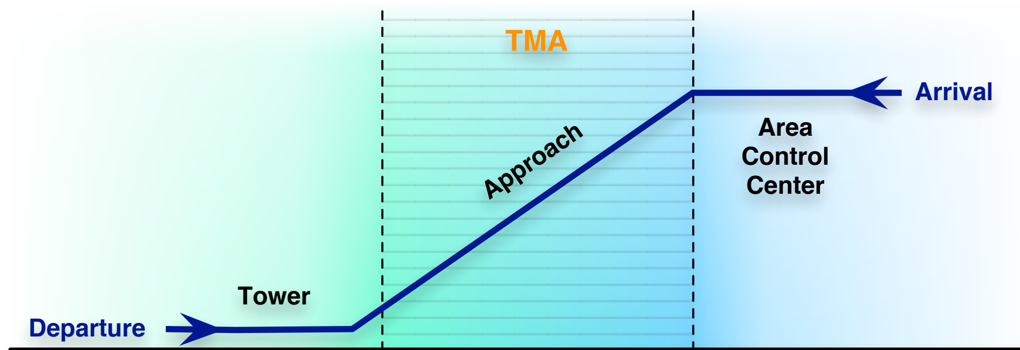


Figure 1.2: *This figure illustrates the responsible units for departing and approaching aircraft.* ***Departure:*** *We see that an aircraft on the ground is controlled by the tower. After take-off, when the plane is airborne, it is handed over to approach control, which hands it over to the area control center once the plane leaves the TMA.* ***Arrival:*** *For approaching aircraft the procedure is reversed. When the plane enters the TMA, the ACC hands the aircraft over to approach, which guides it until the tower takes over for landing.*

during the phases of a flight.

1.3.3 The Departure

In aviation the departure can be explained as the time span from when an aircraft leaves its park position, until it enters the en route phase, which can be sloppily described as the travel phase. First of all, it is very interesting and important to know that basically every action a pilot takes, during departure and arrival as well, has to be approved by an air traffic controller. Most of these steps are done manually via radio communication. The first step, before even starting up the engine, is to call the tower controller and ask for push-back clearance and startup clearance. A pushback is necessary at park positions where the aircraft has no possibility to move away on its own, e.g., when it is parked with the nose targeting the airport building. The reason for this inability can either be that the aircraft does not have the technical ability to move backwards by itself or that it is not allowed to do so because using reverse thrust could, for example, damage the terminal building. The pushback is done via special vehicles who push the plane out of its parking position. Startup means the action of starting the plane's engines. These steps have to be approved by the ground controller, which can be a separate controller, as it is the case at Vienna's airport during high traffic times. On smaller airports or even in Vienna, during times of low traffic, this can also be accomplished by the tower controller who takes over two roles at the same time. Although this differentiation sounds misleading, both types of controllers are in fact located in the tower and this separation into different roles only helps to reduce workload from the people involved. For the same reason, the number of tower and ground controllers varies depending on the size and the traffic volume of an

airport. After starting up the engines, the ground controller instructs the pilot to take a certain taxi route to reach the assigned runway of departure and clears him for taxiing; i.e., tells the pilot to start moving towards the runway. According to the circumstances at the given airport, there can exist a multitude of different possible routes to reach a specific runway, also depending on the parking position of the plane. On big airports with multiple runways it can even be possible that an aircraft has to cross other runways on its way to the assigned one and for this reason all taxi movements have to be carried out with caution to avoid collisions, which, despite strict security measures, still happen every now and then; even on modern airports, like on the 12th of April 2011 when an Airbus 380 hit another plane at JFK International Airport, New York [CNN.com 2011]. For this reason airports usually feature so called stop bars, lights in the ground, which work similar to the traffic lights, people know from their daily life, except that there are only two states, off and red light and it usually is red until the controller decides that it is safe for an aircraft to cross the point. Stop bars often have an integrated automatic mode, which switches the light back to red when the aircraft has passed over it. The taxi phase ends at the holding point, which is the point before the aircraft enters the runway and which is also the latest point where control is handed over to the local controller or tower controller as they are called as well. After the control is handed over, the tower controller is responsible to clear the aircraft for line-up, which means the pilot is allowed to enter the runway and roll to the start position where he receives the takeoff clearance to finally get the aircraft up in the air. Many of the procedures mentioned here have special cases like a direct line-up and takeoff clearance on a free runway where the pilot does not have to wait for further clearance when entering the runway and can directly start to take off. We do not mention all of these special procedures in this work to keep the complexity at the minimum level, to understand the basics of air traffic control.

When the departing aircraft is airborne the tower controller hands it over to the approach control or special departure control in certain high traffic areas. Another term used for approach control is terminal control because it is responsible to handle the air traffic in the Terminal Maneuvering Area (also Terminal (Control) Area) (TMA). This area is a circular area with a radius of typically between 30 and 50 nautical miles around an airport and has also vertical boundaries. This is an area of high importance for air traffic control, since there is obviously more traffic to monitor and handle than in areas where aircraft only pass through and no airport related traffic happens. The special conditions in the TMA are not only a result of the huge amount of aircraft that is often moving inside its boundaries but are also owed to the traffic happening in many different altitudes due to climbing and declining aircraft. Therefore, special care has to be taken to avoid dangerous situations. It is the approach controller's responsibility to guide departing aircraft to an altitude where it may safely leave the approach controlled zone and hand it over to the area control or to a nearby approach control if there is one in the direction the flight is heading.

1.3.4 The Flight

While the plane is in its traveling phase, or en route phase in aviation terms, it crosses different sectors. Sectors are volumes in the airspace created to split it into different segments of responsibility. Sectors are combined into larger Flight Information Regions (FIRs). These FIRs are controlled by another very important authority in air traffic control, the Area Control Centers (ACCs), also known as “Centers”. These are responsible to assign and monitor routing information and altitudes to the pilot to maintain the required separations between aircraft. Separation in this context means the minimum horizontal and vertical distances between two planes to avoid the risk of collision. These minimums differ widely depending on the systems used to track aircraft positions. Separations can get relatively low in areas of good radar coverage but have to be significantly larger in areas where controllers depend on position information received from the pilots via voice communication. Separation is not the only task of the ACC. It also has to respect the routes and destinations of all aircraft in its FIR, which is a complicated task, largely depending on the amount of traffic and potential crossing routes. This task may be further impaired by bad weather conditions. In general ACCs take over flights from terminal control after their departure, or from neighboring centers, route them to other ACCs or hand them over to terminal control when they approach the TMA of their destination airport.

1.3.5 The Arrival

When an aircraft finally reaches the TMA of its airport of destination, it is handed over from the ACC to the terminal approach. Controllers from approach are now responsible that the pilot descends the aircraft to the correct altitude and adjusts its speed according to the following landing procedure. There exist a multitude of different landing procedures. Which ones are available for a given flight is, amongst other things, depending on whether the aircraft is flying under Visual Flight Rules (VFR) or Instrument Flight Rules (IFR), if there is a precision approach possible at the specific airport and on the current visibility. VFR flights always have to land using visual references to the runway whereas it depends on the equipment installed at the airport if a precision approach is available for an IFR flight. The most advanced amongst the systems that allow the execution of precision approaches is the so called Instrument Landing System (ILS), which consists of a localizer system, offering lateral guidance for the aircraft as well as a glide slope or glide path providing the complementary vertical guidance. The main advantage of this approach type, is the fact that the pilot exactly knows his position in relation to the runway he approaches. Non-precision approaches, for example, involve the use of Non Directional Beacons (NDBs) or VHF Omnidirectional Radio Ranges (VORs) where the pilot only knows the lateral position, relative to the position of the mentioned systems.

If an aircraft cannot land at the moment, it will be guided into a holding procedure where it, simply described, circles around a specific point until landing is possible. This procedure can, for example, be the result of bad weather situations or an airport that is currently too busy. If the aircraft is not sent into a holding pattern it will get a runway

assigned where it can land and will receive a landing clearance from the tower controller.

When the descending aircraft is in the final approach and reaches a certain altitude, the so called decision altitude, or a certain missed approach point for non-precision approaches, the pilot has to decide if he wants to land or if for some reason the landing should be aborted in favor of a go-around. The decision altitude is determined by the actual ILS category, which is derived from the current visibility on the runway. If the pilot decides to go around he, again, has to follow a certain procedure, the missed approach procedure, which basically forces the aircraft to climb again and fly along a given route to a point, where another attempt to land can be started.

After landing, the aircraft has to leave the runway as fast as possible through the cleared taxiway and head to its assigned parking position. Basically this ground procedure is the reversed version of the departure sequence and ends with the aircraft reaching its stand.

1.4 Technical Background of Air Traffic Control

This section gives an overview of the different systems used in ATC to support the controller's work while coordinating and monitoring aircraft on their trip. We we will not cover all available sorts of systems but give a detailed overview, which, accompanied by the ATC workflows described in Section 1.3 (Air Traffic Control), highlights the need for this work.

1.4.1 Voice Communication Systems

First of all, voice communication can be seen as the single most important technology in coordinating air traffic in a controlled manner. It is one of the oldest technologies used in ATC but even in times of increased automation, these systems will continue to be important for the unforeseeable future. There are only small portions of the communication between the controller and the pilot that can be replaced nowadays, for example, through the use of automatic departure clearance systems, but most of the procedures in ATC are still coordinated through radio communication. It is the system that, even if many other systems fail, helps the controller to know the position of an aircraft and tell the pilot the next steps to do in order to safely reach its destination airport. These systems are used even in the most remote airports in the world. Even if they only feature a runway, sometimes not even a paved one, they have a radio communication system in place. It is very difficult for automatic text transmission systems or other attempts to do automated air-ground communication, to replace radio communication because of its efficiency in unexpected situations where both parties, on the ground and in the air, have to react and negotiate the next steps quickly. Obviously a direct link by voice is more capable of exposing the feelings or stress level of a pilot, than automated systems.

1.4.2 ATIS

One of the few systems that is in operation at many airports in the world nowadays, to replace the controller's voice, is the Automatic Terminal Information Service (ATIS). This system automatically transmits a prerecorded or synthesized voice message on a certain radio frequency. The message contains important information for arriving aircraft, related to weather, runways, approach procedures and others. The current ATIS message is identified by the use of the letters of the alphabet. The pilot has to tell the controller the current letter, so the controller knows if the pilot is up to date. Anytime something significantly changes, the ATIS is updated with the new values and the next letter of the alphabet identifies the new message. This system reduces controller workload, since the message is standardized and there is no more need for the controller to read the whole message to the pilot personally. Instead the controller only has to assure that the pilot uses the current report.

1.4.3 Positioning Systems

Undoubtedly one of the most important objectives in controlling aircraft is to know where an aircraft is located at any moment. For this reason there exist different systems, which help the controller to keep track of the aircraft's movements. There are, for example, primary radar systems, which can only localize aircraft through their radar signal, secondary radars, which can also identify them through the use of an aircraft's transponder or other systems, like multi-lateration systems, which measure the aircraft's distance to different stations and calculate its position.

These systems are useless for controllers without accompanying software to do a proper visualization of the measured position information. There are multiple steps necessary from the raw input data of the sensors to a visual appearance the controller can work with. First the sensor's output has to be processed for moving targets, which faces multiple problems; from the elimination of ghost targets, which can arise through sensor errors, to the calculation of approximations for moving targets in areas with low signal coverage or behind obstacles. The processing of the resulting data often continues with a data fusion process, which is necessary if different localization methods are in place. Multiple systems can increase the accuracy of the tracking but they also rise difficulties when the results of their initial processing show different realities, due to tracking inaccuracies. Data fusion is responsible to merge these differing situations into the most likely one, based, among other assumptions, on the movements of the target in the previous iterations of the track. Even after a proper track is calculated it is not sufficient to only visualize it on the display because this would not offer enough information of the current situation to the controller. The controller needs a correlation of these tracks to flight plans to be fully aware of what is happening in the area controlled by him. The correlation task can be simplified if systems are in place, like a secondary surveillance radar as explained in [Trim 1990], which already offer an identification of the target.

One of the tasks of such a software system is, to help the controller in detecting possible dangerous situations. For tower and ground controllers for example, it is very

important to be immediately alerted when a runway incursion occurs, which can be greatly enhanced through the use of an Advanced Surface Movement Guidance and Control System (A-SMGCS), as explained in [Piazza 2002]. There exists a multitude of other safety related situations of different severities that have to be avoided, ranging from taxiing aircraft ignoring stop bars, which can lead to collisions with other aircraft, to minimum safe altitude warnings, when an aircraft descends below an altitude where it is safe to fly. Many of these situations can be avoided or at least better detected, through the use of accurate positioning systems.

1.4.4 Weather Systems

Weather can be easily counted to be the strongest influence to air traffic, be it in the air or on the ground. Wind, thunder, ice, clouds, etc. can significantly decrease capacities because of their safety impacts. The threats they pose, range from extended stress levels for the controllers and delayed flights, to forcing whole airports, to shut down flight operations completely, until the weather changes for the better again.

Since mankind has little influence on the weather, it is at least good to know as much as possible about it, to guarantee safe air traffic operations and provide better planning capabilities. For this reason there exist a lot of systems to measure different parts of what can be summarized as meteorological information. The information available to the controller usually contains wind speeds and wind directions at different positions along the runway, covering at least both ends and averaged variations of these values over different timespans. The visibility on different positions of the runway is also measured, called the runway visual range, which means the distance where the pilot has clear visibility. Another important value is the cloud base, which can be either calculated or measured through systems known as ceilometers. Temperature, dew point and air pressure are available as well, the latter one being amongst the most important meteorological information in aviation, as it is used to measure the altitude of aircraft during the climbing and descending phases of the flight. The value is available in different variations, the most prominent being the QNH, which is the barometric pressure adjusted to sea level, used during the aforementioned phases and the QFE, which is adjusted to a specific local reference, like the runway threshold. Some airports are as well equipped with Low Level Windshear Alert Systems (LLWASs), which add an additional level of safety, at airports located in areas where such wind shear phenomena, as described in [Weber and Stone 1994], can occur.

In addition to the information described above, which is directly received by sensors located at the airport, there are different meteorological reports that are generally received through the Aeronautical Fixed Telecommunication Network (AFTN), a communication network that is interconnecting all kinds of different air traffic related entities and is also used to exchange flightplans. The weather related reports are, amongst others, Aviation Routine Weather Reports (METARs), Terminal Aerodrome Forecasts (TAFs) and Significant Meteorological Information (SIGMETs). They contain different kinds of weather information, also extracted from the sensor data available, that are important for ATCOs.

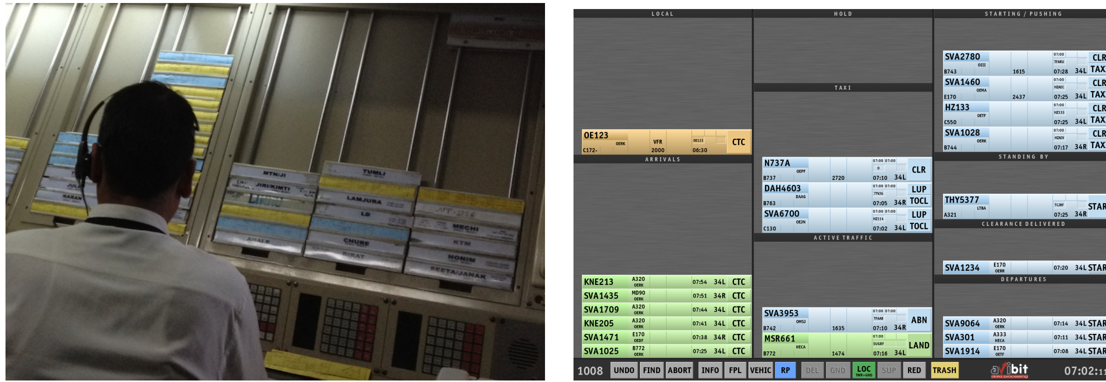


Figure 1.3: *This figure shows traditional paper flight strips to the left, compared to their electronic counterparts, to the right.*

1.4.5 Airfield Lighting Systems

Since taxiways, aprons and runways are also used in bad lighting conditions, for example, during the night or bad weather, they need to be illuminated to make flights possible during these situations. There are many other lights, for example, the aforementioned stop bars or obstacle lights; lights that show certain details of a runway and also the approach lights that are used by the pilot to coordinate the landing of the aircraft he is maneuvering. All of these have to be controlled and this is also done by air traffic controllers. For this reason, there is a dedicated lighting control panel positioned on the controller's working position. This panel is used to turn the lights on and off, change the intensities of some of them and also to change the direction of them, depending on the runway in use. This is needed, so a pilot has only the lights available which are used for the specific runway direction.

1.4.6 ILS & Navigational Aids

The controller also has to know if the instrument landing system, explained earlier in Subsection 1.3.5 (The Arrival) is working. In addition to the instrument landing system the controller also has to know the operational status of the various navigational aids, like NDBs and VORs, that are used by the pilot to navigate the aircraft. If one of these systems fails, it can be necessary to change procedures. For this reason the controller working position is equipped with one or more additional devices to monitor the status of all theses systems.

1.4.7 Electronic Flight Strips

Electronic flight strips are the modern analogy to traditional paper strips used in air traffic control operations. Paper strips are usually printed with some necessary initial information, like the aircraft's registration, also known as the call sign, departure and destination airports and others and are then used, to track the progress of a flight.

Controllers use these flight strips to note all kinds of important information about a flight and to keep track on its current state. Strips are arranged in special boards, which have multiple bays the controllers can use to organize the flights depending on their current state of flight. These paper strips are still common at many airports all over the world but their electronic counterparts are increasingly adapted at ATC facilities, as they keep the physical model of a strip, also respecting the bay metaphor and combine it with modern interaction principles and interactive update mechanisms (see [MacKay 1999]). Figure 1.3 shows a comparison of paper strips and their software equivalent.

1.4.8 Summary

For all the airport related systems described in this section there is one fact in common, their availability at a site differs from airport to airport. Large airports tend to be more complete in terms of having most, if not all of these systems in operation, smaller ones, ones located in developing nations or those situated in remote areas, are often way behind the technological developments and still operate in conditions reminding of the early days of air traffic. This situation sometimes even applies to larger airports, which are just starting to introduce modern systems, often due to a necessary increase in capacity that makes such systems a mandatory requirement to keep operational safety conditions at the required level.

The wide range of different systems makes obvious, why the controller's working position often looks like the examples in Figure 1.1 and why there is a potential market for a system like the one we propose that intends to clean up this working environment.

Chapter 2

Related Work

This chapter deals with the problem domain of designing graphical user interfaces in the context of ATC and the implications of increasing automation in this business in general. There is a large range of aspects that have to be taken into account when creating such systems and a lot of research has been done already in the last decades, serving as a basis for the identification of possible solutions, to the questions and problems we have been facing during this project.

Below we initially discuss the difficulties for Air Traffic Management (ATM) system suppliers when deploying new software to ATC units. Afterwards we examine the strategies on how they can be minimized or at least reduced. The last part of this chapter deals with techniques to improve usability and perception in the development of HMIs that are intended to be used by controllers.

2.1 Difficulties of ATC Automation

Designing systems for ATCOs is a process that can be described as significantly different, compared to users in other businesses, especially when dealing with controllers who have not been used, to be supported by automation software for their work, before. For most HMIs, independent of the domain in which they are used, it is, from a user's perspective, equally important to be designed as intuitively and reliable as possible. But what is considered to be intuitive and reliable is not equal for everyone. Employees in many businesses show a certain kind of skepticism when new software is to be introduced but, as we know from experience in the ATC domain, as well as from the literature, the introduction of new software is accompanied by a degree of skepticism that is a bit higher as the typical office user would reveal in the advent of a new system. Controllers are understandably anxious about loosing control over what they are doing. It is not necessarily the sort of angst that people confronted with an increasing amount of automation typically have, the sort that relates to being eventually replaced by the system. [Kesseler and Knapen 2006] for example, point out that it is not even a goal to go fully automated and remove human beings from ATC, since the human flexibility is considered as a very important skill, when it comes to resolving unexpected situations. Interestingly they point out that this even holds true, despite the fact that investigations found in literature have shown that human error is the most important key factor in aircraft incidents, as well as accidents. According to [Jackson et al. 2000] this phenomenon was also present in the debates of the 80's, where some argued for "the elimination of

human error", while others pointed out the importance of the human controller as "the last line of defense", when everything goes wrong. Considering the current status of automation in ATC, the heterogeneous landscape of these systems, spread around the globe and the extremely low speed, which is inherent to any change in this business, controllers will be needed anyway, for many decades to come.

The skepticism of the controllers is focused on the fact that before having software systems they had to do most of the things manually, like noting flight plans on paper strips or controlling all systems through seemingly trustworthy hardware interfaces. Now they should rely on potentially error prone software systems and, unlike many other businesses, software failures in ATC can lead to incidents or, in the worst case, potentially cost human lives. Reliability is a big issue when new ATM software is introduced and what [MacKay 1999] note, is a spot on comment on what air traffic controllers may think about their familiar, trusty paper strips if they should replace them with electronic strips. In contrast to software, paper strips "are reliable and - unlike computers, telephones, radio, and radar - do not break down". They also discuss another important fact that we have learned from our experience with controllers as well. ATCOs are not generally against the introduction of automated tools, as long as they do not introduce, alongside improvements in the workflow and increased efficiency, new problems or lower safety. Lowering safety also means dangers that are not immediately obvious for non-ATC people at first sight. One of this dangers is, for example, a possible reduction of attention during less stressful times, due to automation, which is decreasing their situational awareness. [Kesseler and Knapen 2006] for example, explain, how controllers train their situational awareness through routine. Doing the same tasks over and over again, even in low traffic times, keeps them prepared for higher traffic situations and the absence of routine can lead to the aforementioned decrease in situational awareness, which can in turn lead to accidents.

[Hilburn and Flynn 2001] also found out that controllers are not generally against the introduction of new tools and that a majority of the interviewed controllers do not mistrust new technology. An interesting detail of their research results, concerning the controller's skepticism against automation, is indeed that, whereas controllers consider the increasing traffic volume as the biggest upcoming threat, it is the management that predicts the increasing automation in ATC to be the biggest safety issue. The concerns that came up during their focus group interviews mainly include the shortage of personnel, combined with increased traffic volumes. People expressed the need to specify the requirements of new systems as precise and early as possible and demanded the involvement of controllers during specification and development. They also claimed the demonstration of the benefits of new systems and requested sufficient training for these systems and the workflows they implicate. These findings result in some very important implications that also correspond to our own experience. When developing systems that should be accepted by controllers, it is important to involve them in the best possible way and how to support this as good as possible, is definitely one of the key focuses of the framework we present in this work. Engineering ATC applications is definitely different from other businesses, where software is often bought without considering the user's needs and where they just have to use it, whether they want or

not. As [MacKay 1999] emphasize figuratively, "air traffic controllers have a real voice in the technology they use. Because of the safety-critical nature of the system, they can reject interfaces they do not like. Controllers have the final say for a very simple reason: if there is an accident, computers do not go to jail, controllers do."

2.2 ATM System Developer's Survival Strategies

For software engineers the attitude of controllers, presented above, results in the challenging mission to create as much trust between the controller and the delivered piece of software as possible. At first, the controller should get the feeling that he can rely on the system, without having to fear of potential data loss, of not having all the information available he would have had before the introduction of the system or of loosing any functionality he had, when working with paper. And second, it is very important to provide the best possible user interface experience, in terms of keeping situational awareness as high as possible. For these reasons in the remainder of this chapter we discuss strategies we found during our research, on how to improve the process of development, specification and delivery, as well as the design of ATC systems, focusing on the HMI part.

2.2.1 Bringing the Controllers on Board

How is it possible for developers to take the fear out of automation and convince controllers that the system they will be delivered is beneficial for them? First and foremost this promise should be true. As mentioned before, controllers are not the kind of users that will accept a product that is of no use, since this would only mean changing workflows for no additional benefits and potentially increased threats on safety.

In the context of building trust, [Hilburn and Flynn 2001] deliver very useful examples of best practices, on how to accomplish this important goal. At first, it is a very good idea to involve the controllers, as early as possible, in the development to share information about the planned system with them and let them share their working procedures as well. Regular prototypes, right from the beginning of an automation project, can be of great help in demonstrating future benefits to the controllers. In turn, developers can get regular feedback and learn to better understand the controllers' needs. This process assists in creating a human centered design, instead of one that is driven by the engineer's understanding of the technology behind it or of how things should work. [Kesseler and Knapen 2006] mention another important function, prototypes can fulfill; they can greatly help to specify the exact requirements for the system, something that is often very difficult or even impossible at the beginning of such a project. And the iterative nature of the prototyping approach helps in tightening the specifications along the project life cycle. It can also be helpful in identifying usability problems and performance issues ([Leonidis et al. 2012]). [Mertz et al. 2000], however, point out that the prototyping method, even if the authors feel confident about its efficiency in general, has to be executed with caution, as there exist many examples where prototype evaluations from users, led to systems that were not really usable when running in an operational setting

later on. This implies, prototyping and the evaluation of its results, have to be done carefully and it is probably a good idea to involve controllers who already have experience in the use of ATM software.

Prototypes are not only useful during the development of new applications but, even if Commercial Off-The-Shelf (COTS) systems are going to be installed, there will always be customization needed, at least to a certain extent. This is due to local specialities and, as [Jackson et al. 2000] emphasize, controllers are probably not "willing to accept a 'second hand' HMI". The requirements for these adaptations can be gathered during such a prototyping session or workshop. Additionally this can, again, serve as an impulse for the controllers involved, to build a feeling as if the system was designed by them.

But even before any prototype is available, the interaction with controllers, for example, in workshops, is a very important and helpful experience for all sides, the controllers, the engineers and the management as well. From our experience projects always work better, if all of these parties agree on the decisions made and none of them is left out or overruled by another. This also leads to a common understanding of the requirements and technical possibilities and shifts the perspective away from a customer supplier relationship, to a productive partnership.

[Hilburn and Flynn 2001] also conclude that in the later stages of development and after delivery, trainings are another crucial factor to further support the controllers, to gain confidence in the new system. Furthermore, the authors state that it is important for engineers to also have enough domain knowledge, to gain credibility when communicating with controllers. Although we agree that this shall never be underestimated, from our experience, developers should never expect to be as respected by controllers as other controllers are. This is for obvious reasons: No developer, unless, he was a controller before, can learn all the operational details a controller has in mind when thinking about his workflow. Engineers can show the controllers how things can be done with the system and what is possible technology and effort wise, but they should not try to tell them, how they will have to do things in the future, since this will in most cases not work and destroy much of the efforts invested before, in building trust.

2.2.2 Providing the Best Possible User Experience

Now that we know that we have to involve controllers and that we have to show them real benefits in the system, their management bought or wants to buy, the question remains, when a system is beneficial for controllers? If we would have to mention one single most important challenge for controllers and as well one of the most recurring topics in ATC automation related literature, it would be situational awareness. In the end, every decision a controller has to take, every conflict he realizes and resolves, depends on his ability to be aware of the current situation.

If we consider situational awareness as one of our ultimate goals, what kind of implications does this have on the design of a software system?

Let us briefly summarize the situation of an ATCO: The controller needs every information in real time, and he has to observe different systems at the same time. So, we cannot rely that a controller is even having a look on our system when a special

situation occurs. Depending on the situation, controllers might not have the time to browse through an endless chain of cascading menus to find the information they need, because at the same time they may have to check a multitude of other things. They could have to note something on their flight strip, keep an eye on the radar screen and, not to forget, they have to communicate to the pilot on how to proceed. One may argue that controllers are trained for this kind of situation and they have managed them before automated systems assisted them in their day to day business but, on the other hand, it would be naive to assume that management only buys these systems solely to make the controller's life more comfortable. These systems are bought to increase the capacity of an airport or an airspace, to let the controllers handle more aircraft in the same amount of time and to reduce the separations needed between aircraft, when landing or departing. The increased capacity can suddenly fade away if one of the systems, that should assist the controller, fails or an otherwise unexpected situation occurs. All of a sudden the controller will have even less time to handle these situations than he had in the times before decreased separations and increased traffic.

Aside from the need for total stability and reliability of ATM systems, it seems that when designing user interfaces under the premise of optimum situational awareness, one has to walk down a very thin line between showing as much information as needed but on the other side, only when it is really important to the controller. It is easy to flood the user with information that may be irrelevant for the time being, but can become very important under certain circumstances. The potential information overkill raises many design questions, since it can lead to reduced situational awareness of the controller, due to potential mental overload, which would again be a safety risk. The danger and potential safety impact of missing the sweet spot between showing too much and too little information is also described by [MacKay 1999].

2.3 User Interface Design in the Context of ATC

There are many different ways to communicate information apart from bare sensor data or status information to the user and especially in the ATC domain a lot of research has already been done concerning these methods. Even if most of the available literature is related to improving the way controllers perceive changes in flight data on electronic strip systems or situation changes on radar displays, the lessons learned can also be applied to a system that tries to enhance the user experience for secondary systems, like it is the goal for the framework we want to create. We briefly summarize the possibilities to transport information to the user in the following subsections, including the advantages and possible disadvantages of the listed methods.

2.3.1 Colors

One of the oldest and most well-established ways to present information is the use of colors. The likes of [Bertin 1983; Tufte 2001; Ware 2012] already point out the importance of color in data visualization and show that a good visualization makes careful use of it, instead of using an excessive amount of different colors. [Graham 1997] also states that

color is an effective way to show priorities and mentions guidelines on their use, but, in contrast to the former mentioned authors, in an ATC related context. Through the use of different colors, special information can be better distinguished from general data and the user's attention can be drawn more easily through the highlighting of special or currently selected items. Large blocks of color should be avoided as well as colors that only differ slightly. In general no more than eight different colors should be used to transport information, which is reasonable since otherwise it can get overly complicated for the user to memorize the different meanings. Also colors with well known meanings, like yellow, which is often used for warnings or red, signaling alert or danger in many cultures, should not be used outside of these use cases. In [Cardosi and Hannon 1999] the use of colors in ATC systems is also covered extensively. They conclude that the first important decision, before creating a color coding scheme is to select a proper background color, which is, as they suggest, a light grey in light environments and black or a dark grey in darker, bad lit environments. The better the selection of the background color, the better it supports the distinction of the different colors chosen. Contrast is very important, contrast in luminance even more than contrast in colors (see [Ojanpää and Näsänen 2003]). Also in contrary to the former mentioned eight colors, they propose a maximum of six different colors that should be assigned different meanings. Next to these important guidelines, a large focus in their research lies on the many traps that can be overseen when using colors. If no attention is paid to these traps, the wrong usage of colors can easily cause more harm than bring advantages. They also mention that animations, e.g., blinking, are better suited to produce fast perception than changes in color are. Wrong or overused color-coding can reduce perception of information significantly. Differences in color perception amongst people should as well not be underestimated. To be not confusing, color has to be used consistently, in the ideal case in all systems present at the controller's working position. This is unfortunately rather difficult to realize, since there is a multitude of different suppliers for these systems; all of them providing their own, different solutions. This a problem an integrated system like the one we propose in this work can solve partially by reducing the number of different approaches to HMI design.

Even tough there are lots of errors that can be done if colors are used in the wrong way, they are generally considered as being a good method to present information on ATC HMIs, especially if tested thoroughly, ideally in the target operational environment of the system, using the target display hardware.

2.3.2 Animation

As already mentioned while discussing the usage of colors, animations may be better suited to indicate changes to a controller than "static" color changes and can supplement the user experience. More and more Graphical User Interface (GUI) frameworks make it relatively easy to include animations, so user interface developers do not have to stick to the sole use of color coding anymore, when developing new applications. During this chapter we have already mentioned that controllers may not be looking at the display, when important information arrives, due to the fact that they have to observe multiple

displays at the same time. Enter Animation. Research of [Athènes et al. 2000] focuses on the ideal design of alarms and notifications, since their immediate perception is a safety issue and missing them can be lethal in the worst case. They approved that animations are an efficient way to capture a human being's attention because they are able to stimulate the peripheral vision. Peripheral vision reacts to movements very well and activating this part of the human's visual perception is exactly the goal we have to meet when designing an application framework for a system, that most of the time is not in the user's central view. In a study they measured the influence of certain parameters on the reaction time of their subjects and their results show, that the more transparent a signal is and the slower the change of the signal, the longer the user needs to react to it. It also has an impact, if the signal is locally isolated or is more globally visible, the latter one leading to lower reaction times but only if the signal is very transparent. So the more opaque, step-like and global a signal is, the faster it is perceived. From these three influences the authors consider transparency as "the trickiest, but the most promising" one. These results should not be ignored when introducing animations into an HMI. [Schlienger et al. 2007] also address the use of animations and highlight their use in increasing perception and, therefore, improving situational awareness. [Mertz et al. 2000] also point out the advantages of animations, which can be used to notify events or state changes through animated transitions. They can as well be used to simply improve feedback from the user interface when used for opening or closing menus, or as well, during the now well known kinetic scrolling, everybody knows from touch enabled devices, which are ubiquitous nowadays. They conclude that the use of animation is a very efficient way to provide feedback to the user.

2.3.3 Fonts

The selection of appropriate, which means first and foremost legible, fonts is important for any application, independent of business domain or output device. This is obvious to anybody who has ever used software or simply read a book or newspaper. What is not so obvious, is the difficulty it can pose to find such an appropriate font. A font that is well suited for one purpose, can be horrible when used in a different context. Should it be a monospaced font, should it have serifs, should it be bold, italic and so on. All these different styles have their use cases but the art is to find the correct styles for a use case. Next to the obvious duty of fonts they can also be used to transport other information than just the words and values they represent. While also discussing the use of fonts in general, [Mertz et al. 2000] propose a very creative use of fonts that does not seem obvious in the first place but is an interesting option when trying to transmit meta information about the data to the user. They use different fonts to let the user easily distinguish between data that is entered by himself or his colleagues and data calculated by the system. The user input is presented through a well chosen font that resembles of hand writing, whereas the data received from the system is presented by a "computer font". A meaningful, deliberate choice of fonts is very important. From our experience, we know that in ATM systems it is, for example, relevant to be able to easily distinguish a zero from the letter O, which is not accomplished by many of the fonts available. This

is only a small example, but in ATC, the controllers often have to transmit messages or values to the pilot and since they tend to speak fast, they should also be able to read it fast.

2.3.4 Shapes

Another way of transmitting information to the user is the shape of an element. An arrow, for example, can be used in many cases. It can indicate the wind direction on a wind rose, whereas on a flight strip it can indicate the direction of the flight. Different shapes might as well be used to indicate different priorities as mentioned by [Graham 1997] or to indicate outstanding or completed tasks through the well known empty or checked boxes, which are an integral part of most GUI toolkits. The experimentation with meaningful and easily identifiable shapes can be a profitable investment, especially considering how easy and efficient it is, to recognize the meaning of pictograms, like, for example, the one that is used for arriving and departing aircraft at many airports in the whole world.

2.3.5 Sounds

Sound is one of the most powerful aces a system can have up its sleeve. The positive influence of sound in user interfaces is, for example, assessed by [Schlienger et al. 2007]. But at the same time it is a double edged sword. From our own experience we know that sound has to be used in small portions. We tend to not use sounds for anything other than to signal alarms. Systems that tend to make an overuse of sound notifications, even if the controllers requested most of these notifications at first, run the risk of annoying the user, instead of helping him. The consequence is the user turning off most of these audible notifications again, to get rid of the noise pollution. [Athènes et al. 2000] also mention the preference in ATC environments, to keep the use of sound at a minimum and reserve it for emergency cases. [Cabrera et al. 2006] as well, describe the influence of frequent audible alarms on the controller, which leads to “desensitization and annoyance” even if the assessed sound alert scheme itself is generally perceived well.

2.4 ATM System Development

The problems mentioned in this chapters are not easily comparable to other problem domains as there are many aspects in this domain that create very particular difficulties, like the fundamental requirement to keep situational awareness high.

As an interesting side-note, we have to mention that from our impression the ATC automation related research in the last decade, was heavily focused on the algorithm and optimization side, leaving only a minor emphasis on HMI related topics as can be seen, for example, when browsing through the proceedings of the [ATM n.d.] over the years. This impression leaves us even more interested in the usage of modern frameworks and technologies and is, hence, one of the reasons for this project.

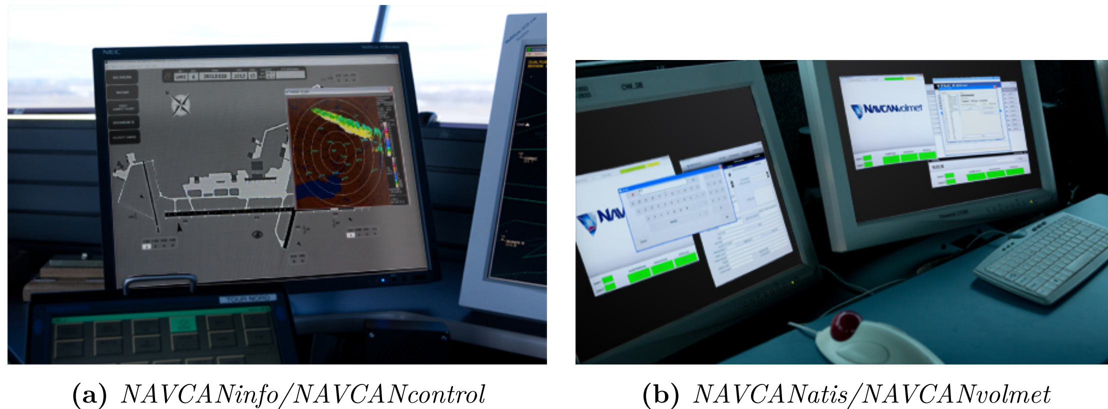
One of these new technologies can, for example, be seen in the changing idea of desktop metaphors. There is a shift in user interface design, away from classical "Windows, Icons, Menus, Pointer" (WIMP) interfaces, to pen or touch based interfaces using more direct methods of interacting as [Mertz et al. 2000] already mentioned; next to the lack of available literature covering the modern possibilities at the time of their research. Over a decade later this situation has still not changed much, aside from the occasional research on electronic strip systems. They describe the former interface paradigm as consisting of a "plain and strict appearance, coarse interaction styles (...), indirect interaction through rigid devices (mostly a mouse), absence of visible feedback, heavy use of menus and windows". Touch based systems and their more advanced graphical user interface frameworks allow for much more flexibility in the presentation and handling; for example, the use of gestures. One of the advantages of touch based interfaces, the authors point out, is "that the user may be able to interact with less visual attention in a semi-blind mode. He has to look at the screen where the target is located, and then he can point his finger on this target without tremendous visual attention", which leaves more time to do other things than tracking a cursor until it reaches the final destination of interaction. These interfaces also introduce issues that have to be addressed, like the reduced precision of a finger, compared to a mouse pointer, which logically has to result in larger interactive components but as the authors mention, it may as well introduce totally new capabilities, like having two controllers work on the same screen, which would otherwise be a horrible idea, considering that they would have to share a single mouse. [Conversy et al. 2011] for example, describe a collaborative system that makes heavy use of this idea.

Finally, there are different techniques to improve the process of introducing ATM systems on the one side and optimizing the delivered GUIs on the other side. In the end, every decision made when designing a user interface for ATCOs has to be evaluated against its influence on the safety implications it has. More often than not, these influences can only be judged by the people who probably know it best, the air traffic controllers themselves.

2.5 Existing Tower Information Systems

Before closing this chapter, we present examples of existing integrated information systems that are available on the market nowadays. Since these products can usually only be seen in operational ATC environments or at trade shows, we have to use resources from the web to show sample images. Since the retrievable information about these products is primarily marketing material, we can only make assumptions about the real functionality of these systems and the technology they are based on. Therefore, we mainly present pictures of the products, add some features that the companies claim to provide and comment on our impressions about the product.

Figure 2.1 shows an information system provided by NAV CANADA, a Canadian ATM software provider. According to their web site (see [NAV CANADA 2013]), the system, named *NAVCANsuite*, is capable of providing weather information, airfield light-



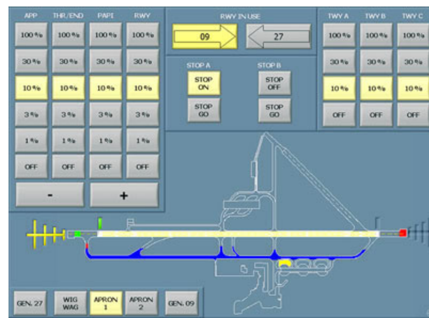
(a) *NAVCANinfo/NAVCANcontrol* **(b)** *NAVCANatis/NAVCANvolmet*

Figure 2.1: *The pictures show components of an information system provided by NAV CANADA.*
Copyright by NAV CANADA (see [NAV CANADA 2013])

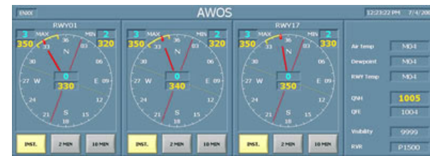
ing status and control, as well as the display of charts, maps and others. There is also an ATIS system available and other components. From the images, we assume that all these components, although claimed to be an integrated system, are separate pieces of software, not built upon a common base and not offering a homogeneous user interface experience.

In Figure 2.2 we can see an integrated ATC information solution made by ACAMS AS, a company based in Norway (see [ACAMS AS 2013]). On their web site they state that the system integrates airfield lighting functionality, meteorological information, navigation aids status monitoring and control, charts and many others. On the screenshots we can see that the system seems to make use of very tiny fonts and the GUI technology in use looks rather outdated. Everything seen on the screen, seems very crowded and lacking overview.

Finally, in Figure 2.3, we see an information system developed by Frequentis, a company based in Austria (see [Frequentis 2013]). The product suite, named *TapTools*, seems to offer functionality for a wide range of ATC related systems. Although the presented software components look rather modern, especially compared to the previous example, they seem to provide rather crowded and low contrast layouts.

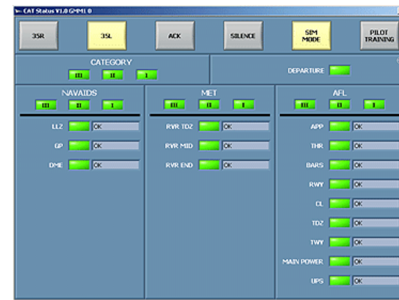


(a) *Airfield lighting component*



(b) *Meteorological information*

(c) *Navigational aids component*



(d) *Combined component for airfield lighting, navigational aids, and meteorological information*

Figure 2.2: *The pictures show components of an information system provided by ACAMS AS.* ***Copyright by ACAMS AS (see [ACAMS AS 2013])***

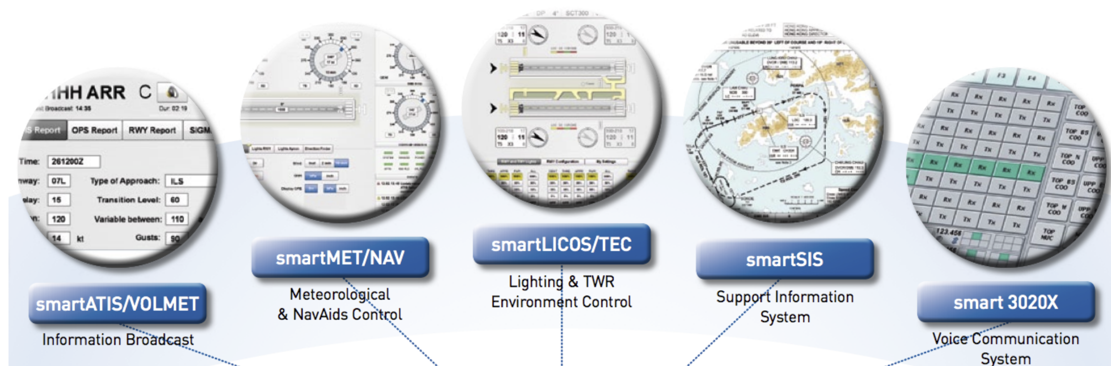


Figure 2.3: *This figure shows pictures of different components of the TapTools Suite offered by Frequentis.* ***Copyright by Frequentis (see [Frequentis 2013])***

Chapter 3

Methodolgy

This chapter outlines our methods and approach on the realization of the project goals. We explain how we have solved different problems we were facing and describe the different critical steps, leading to the final solution presented in this work. What we present in this chapter serves as a road map and gives the reader a summarized overview of the used methods and steps taken on the way to our final design decisions.

3.1 Requirements Specification

After finishing the research on ATC specific user interfaces we have developed a concrete idea on how the system should look like and the feature set it should support. The idea is to create a hybrid system. On the one hand it shall build the basis for a robust application that can be delivered to customers and on the other hand it shall be capable of serving as a rapid prototyping framework.

In the beginning we planned to develop a modern user interface, incorporating most or all of the best practices in User Interface (UI) design that we have found in the relevant literature, discussed in Chapter 2 (Related Work). However, we soon have realized that this is not what we really want. For this reason we shifted our goals, away from the unrealistic idea to create a system that brings arrangement of HMI components, visual appeal and usability to perfection. The shift we have made was towards the creation of a framework that allows to flexibly experiment with different user interface concepts. A framework that facilitates the easy adaptation of new developments and results of user interface design research; one that can be fully adapted in terms of the HMI, without necessary changes in the fundamental system.

After getting a precise picture of what we would need to achieve with this work, we have defined the requirements for the resulting system, incorporating AviBit's wishes as well as our ideas.

3.1.1 Requirements and Goals

The basic requirement for this project is to develop a client-server based framework that can integrate different data sources and provides flexible means, to adapt it to different HMI requirements.

In the following paragraphs we deal with the requirements and goals of the presented framework as a result of the specification phase. As mentioned before, these require-

ments are a combined result, on the one hand defined by AviBit and on the other hand implied by our own research and the lessons learned from studying the available user interface related literature. From AviBit's side there are constraints in terms of the general requirements predetermined by their tool chains and frameworks in use. ATM systems are complex systems that also have to fit into a certain environment. Like already shown in Chapter 2 (Related Work), the ATC business is rather rigid and this characteristic is probably inherent to many businesses, where every change of technologies or processes that are working reliably, means the possible introduction of safety issues, or things working worse than before. The same comprehensible conservativeness applies to AviBit as well because proposing a system or technology that completely changes their technological environment, would carry the large risk of introducing unknown consequences concerning reliability, testability and maintainability. Therefore, the overall goal of the system is to deliver the best possible result, based on what we have learned from dealing with the research topics, without disrupting the technological landscape of AviBit more than needed.

The subsequent part of this subsection contains a list of objectives for this work. It contains requirements and goals alike. We split these into three different categories to emphasize their different context. The general requirements are related to the overall functionalities, we want to achieve with the system, while the technical requirements are mainly a result of AviBit's existing environment. The third category, the user interface requirements, shall reflect the possibilities the system shall offer, to create sufficient HMIs. The order of their appearance, however, does not reflect their importance.

First of all we present the overall goal of this work, the one that stands above all requirements and that in the end is the one we will evaluate for its success. At the end of the list, we discuss the declared non-requirements or non-goals respectively to clarify what should not be achieved with the presented framework and to setup a boundary line to other ATM systems.

- **The Overall Objective**

  As a preface to the goals and requirements listed below we formulate a meta goal for this work, the goal that defines the overall achievement in which this work should culminate and that shall be the basis for the evaluation of the results.

  The system shall consist of a robust framework and offer basic implementations of different system components, while leaving a lot of room for further experimentations with presentation techniques and the possibility to easily integrate them into the system. This shall result in short prototyping cycles and the possibility to try out more different versions of new ideas and gather more feedback from controllers.

- **General Requirements**

  - **System Integration**

    The main goal of the proposed TIS, is to offer an easy way to integrate the functionality of other systems. The integration shall focus on secondary systems, including many of the systems in Section 1.4 (Technical Background

of Air Traffic Control), like weather systems including sensor data, status of sensors, as well as weather reports, airfield lighting monitoring and control, navigational aids status and others. Additionally it shall be possible to present different charts and documents available at an airport to the controller, offering simple document retrieval functionality. Document management shall not be part of the software, as this shall be done by other means. Although we call it a TIS, it should as well be suitable to be deployed to other ATC-related units, not necessarily located in the tower, as, for example, area control centers.

- **Flexibility and Extensibility**

  Prior to asking for consultancy in the development of a concept for a new future-proof solution, AviBit has already developed a previous version of a TIS. It was their first step on the way to be able to offer an integrated solution as described in the previous requirement. This system, however, did not fulfill their requirements and missed the target for a number of reasons:

  - Initially the system was intended to be configurable to a certain extent but due to the following circumstances the system was more or less tailor-made for one specific customer, where the system was deployed to only one site:
    - Qt3, the library that was used to implement the system, was rather old and inflexible and already out of its lifetime, long before the development started.
    - In the final development phase, in order to meet deadlines, development was rushed and initial plans to strive for configurability, were abandoned.
    -
  - The system was deployed to only one additional airport in another project and the adaptation times, needed to realize this, proved to be very bad.
  - Many workarounds and tricks were needed in the code to adapt the system.

  The above reasons can be seen as an evidence for the misconception of the original system. The adaptation of the system to the mentioned additional project, will be taken into account, when evaluating the outcomes of this project in Chapter 5 (Results).

  To avoid these mistakes of the past, one of the most important requirements for the system is that it shall be easily adaptable and extensible. This shall be achieved through a high degree of configurability and a framework that delivers a solid foundation, while at the same time allowing for the easy addition of new features and components, without having to alter many parts of the existing system, i.e., the core framework.

For the execution of new projects, the development of interfaces to other systems should be accountable for most of the efforts and time spent because this part won't be avoidable, unless system providers stop using proprietary protocols. However, the adaptation of the rest of the system shall be reducible to a minimum and only consist of configuration tasks, including layout adaptations of the HMI and the occasional implementation of new graphical components to display information not needed before.

- **Rapid Prototyping**

  In Chapter 2 (Related Work) we discuss the general importance of building trust between the system provider and the end users, the controllers. AviBit's experience in its past projects already helped them to develop an efficient process of defining requirements and functionalities. This process often includes exactly what we have learned from our research, namely to work out these requirements together with the ATCOs, who in many cases are not even aware of their requirements themselves. This unawareness often is a result from not having any experience with modern computer-aided air traffic control solutions. Therefore, AviBit has started to work closely together with the customer on the one hand and ATC-related consultants with long-term experience in using such systems, on the other hand, to get the most out of the delivered system. Having well experienced controllers available, when talking to future users, which are new to the system, accelerates the design process even more than engineers that have good domain knowledge. In most cases the best results are achieved during workshops were it is very helpful, to already have a prototype available, as we have mentioned earlier. The prototype should to a certain extent already reflect the final operational circumstances at the site in question. In the past these workshops were often held with the use of composed imagery or mock-ups, which are of limited use when it comes to the goal of delivering a feeling for the final product to operational staff, who want to see the system in action to better understand its usefulness and operational impact. Therefore, another requirement for this project is that the final system shall deliver the effortless creation of prototypes for such workshops, which not necessarily have to be thrown away afterwards but shall already be the basic configuration of the final product. This means the prototypes as well as the final product shall be derivable from the same framework to reduce development time spent, in the realization of projects.

  Having a prototype available in the early stages of a project and skipping the image mock-up phase can also be a cost factor, since it can reduce the number of workshops needed to define the system.

- **Technical Requirements**
  - **Qt Library and Platform Independence**

    A very important requirement for AviBit is that the proposed framework

shall use the Qt library, which is discussed in greater detail in Chapter 4 (System Overview, Design and Implementation). A requirement that has a big influence on the design and development of a new system, since it limits the possibilities one has, in evaluating target-technologies. The required use of this library is a result of AviBit's experience with it for years. Every piece of software in AviBit's portfolio uses the Qt library to stay platform independent. This is necessary, since even though the majority of the delivered systems are running on Unix derived operating systems, there are cases, where it is also necessary to support the Microsoft Windows platform or others. Customers, for example, might formulate the requirement, to integrate the system into their already existing environments. For this reason it is not feasible, to rely on platform dependent libraries. Another reason to continue using this library, is the aforementioned established technological ecosystem. Using another library, as the foundation for this system, would multiply the efforts needed; for example, to build a solid testing environment. This increase in workload would lead to additional human resource requirements, to maintain the additional dependencies. This shall be avoided.

- **Client-Server Architecture**

  It is also required for the framework to fit into AviBit's typical process and network infrastructure, which again is targeted on an optimized usage of existing procedures and technologies, related to testing and maintenance. The design, therefore, has to respect that there shall be a client-server architecture. Additionally the server shall offer a sufficient communication module to be used to communicate with special dedicated data acquisition processes that are used, to communicate to external interfaces, e.g., external systems that deliver sensor data.

- **Thin Client**

  The client shall be kept as "thin" as possible and act mainly as a data representation facility, handing over most of the logic to the server and its connected data acquisition processes. This concept is intended to further improve the generic nature of the client to fit different project setups.

- **User Interface Requirements**
  - **Custom Components**

    Considering the user interface, it shall, first of all, be possible to implement custom graphical components in an easy way that is not limited by the framework.

  - **Homogeneous Look & Feel on All Platforms**

    Another necessity is related to the technological platform independence requirement, defined above: The out-coming framework shall be able to offer a user interface that has the same look and feel independent of the target platform, where the final HMI will be running on. This shall help to offer a

unified user experience within the application, irrespective of the underlying operating system. Therefore, it shall be easily avoidable to get trapped by platform specific behaviors or graphical implications, resulting from different underlying window toolkits.

  - **Applicability of User Interface Design Principles**

    In order to be able to continuously improve the user experience of the system, the framework shall be capable enough to offer all necessary graphical functionalities, to experiment in the research topics covered in Chapter 2 (Related Work).

  - **Touch Functionality**

    With the advent of touch-enabled devices, even in air traffic control environments, a new approach to user interface design is very welcome. The use of these devices can greatly enhance usability, an effect already discovered during AviBit's development of a paper-less flight strip system. These systems in general are not based on input through finger gestures but on pen-like input devices because, often times, it is necessary to quickly note down additional information on such a strip. This is best done with a pen, as the controllers were used to this technique for decades, while using paper strips. Nevertheless, it can be a big benefit in stressful situations, to not have to grab a keyboard or mouse, to get to the needed information from a supporting system. It would be more comfortable to only trigger an action, through simply hitting a button with a finger or pen-device. For this reason it is a requirement by AviBit that the developed framework shall allow for easy integration of touch or gesture based input methods, next to traditional desktop behavior. The latter cannot always be easily avoided due to the customer's operational requirements or spatial circumstances in the controller's working environment, where there may be not enough space to install a touch device in a comfortably reachable position.

- **What it Should Not Be**

  As research shows there exists a large amount of different possibilities to improve but also to impair the usability of HMIs and it is practically impossible to implement a system that does all of these things right, especially in the scope of this work. For this reason the final result of this work shall not be a system ready for deployment to airports but a solid foundation to build on.

  While the focus of the framework shall lie on system integration, it shall not process, store or display any radar data or positional information, gathered by other location technologies, like multi-lateration systems, since, aside from the data processing part, it would require different user interface metaphors than the ones that shall be applied to this system. The same applies for flight information. The developed system shall not try to present flight plan information for individual flights in special ways, i.e., shall not be a replacement technology for paper strips,

electronic flight strips or alternative methods to process or interact with flight specific information. The proposed system shall serve as a complementary system, filling the gap between the other two mentioned types of systems.

3.2 Technology Evaluation and Decision

Before starting with the design of the system we have decided to plan an extended evaluation period, to assure that the selected toolset will help us, to realize the goals set for this work, as defined in Subsection 3.1.1 (Requirements and Goals).

Basically this phase consists of a lot of prototyping, experimenting with the technology chosen for evaluation and finding out what is possible, what is not and if the possibilities to support our ideas, outweigh possible disadvantages.

During this evaluation we have found a solution that fits our idea of an application, like the one we outline above. A framework that on the one hand supports all modern ways of user interface design and on the other hand, allows for very easy, though powerful configurability and extensibility. Coming initially from the need to create a modern Application Programming Interface (API) for the mobile application world, it supports most of the possibilities currently desired, from animations, states and transitions to touch screen ready input methods that are easily usable on desktop systems as well.

The detailed results of the selection, especially the reasons that legitimate the chosen toolset for the planned purpose are discussed in further detail in Chapter 4 (System Overview, Design and Implementation).

3.3 System Design

The design phase is a crucial part in the success of a newly developed software product. We have invested a lot of time to create a design that meets the requirements and goals we have set up before. Obviously every part of the system has to be well designed. For a distributed system it is not sufficient to only focus on the HMI part, even if it is the main focus of this project. For this reason, during the design phase, we strove to create a very flexible design for all involved modules. In parallel to this goal we have also emphasized to keep the design as simple as possible. These are the two main design principles we have been following for this framework; flexibility and simplicity. The design process itself was an iterative one because during the implementation, we have rethought details of the design concept a few times. This has never resulted in very drastic redesign decisions but tweaks have been needed from time to time if new ideas and functionalities had to be incorporated that had not been clear at the beginning of the project.

3.4 Evaluation Method

We already have explained the goal we have set for the framework we present in this work. On the one hand it shall be able to offer rapid prototyping capabilities and offer

flexible possibilities to extend or change the user interface and on the other hand, the created user interfaces shall offer a distinct user experience that is accepted by the air traffic controllers that will have to use it in the end.

There is one big difficulty that is raised by these goals, namely the question on how they can be measured or tested. It is difficult to put them into concrete numbers, since all of them are fairly subjective goals. Nevertheless, we need to have an indication on the success or failure, of our efforts spent in this project. In the following paragraphs we explain our approach on how to evaluate the fulfillment of these goals. The results of these approaches are discussed in Chapter 5 (Results).

A reasonable approach, to answer the adaptability question, is to measure the time it takes do deliver a customized prototype and to realize changes that are discussed during the presentation of such a prototype. For this reason we test these capabilities by tracking the times spent on adaptations during the time of this project. This gives us the necessary numbers to objectively evaluate the framework's suitability to deliver projects as fast as possible.

The controllers' acceptance of the system presented to them, is the second important indication for the frameworks suitability for its purpose. Since the system is not yet operational at any site, we cannot do long term studies of the usability and acceptance of the framework. The way for us to get feedback from controllers that we can evaluate, is, to present them the aforementioned prototypes during the design workshops they have with AviBit. During these workshops we gather impressions and comments of the controllers and select the most important comments, to discuss them together with the presentation of the prototypes in Chapter 5 (Results).

Chapter 4

System Overview, Design and Implementation

This chapter addresses the framework created during this project in detail. It is divided into three main sub-chapters. The first part deals with the library that is used to build the core of the system and includes a detailed description, of the evaluation we have done, before making a final decision on the technology. This technical description outlines the possibilities and limitations of the framework, as well as the reasons for its selection.

The second part is about the design that we have created. It provides a high-level overview on the separate modules and their functionalities.

Finally, the last of the three sub-chapters deals with concrete implementation details on the developed system.

4.1 Technology Overview and Evaluation

This section provides a general introduction to the Qt library, and the important QtQuick submodule that we have used for the system and an extensive report on the technology evaluation that we have done on this submodule. For the evaluation of QtQuick we wanted to

- learn about the general features of the module and if it fulfills our requirements concerning user interface creation;
- investigate the possibilities to extend the module with custom components;
- investigate the feasibility to create complex applications with QtQuick that use C++ as the backend;
- investigate the feasibility of an additional GUI editor to create QtQuick HMIs with the developed framework.

4.1.1 The Qt Library

The Qt library is a platform independent C++ library. It provides powerful modules for many of the problems, developers are facing when creating software. Qt provides high-level classes to easily handle containers, networking, graphical user interfaces and

others. OpenGL and JavaScript are supported as well. This library delivers the core of all systems delivered by AviBit and is therefore, as already mentioned in Subsection 3.1.1 (Requirements and Goals), mandatory for the presented system. Note in this context that, at the time of this project, AviBit has just started the transition from the very outdated version 3 of the Qt library, to version 4, which we use in our system. This results in the fact that this project has transferred a lot of knowledge about the modern version of Qt to AviBit and the resulting system shows, how different GUIs can be built, compared to the past.

4.1.2 The QtQuick Module

QtQuick is one of Qt's modules and is a framework to create graphical user interfaces through a declarative language called "Qt Modeling Language" or "Qt Meta Language" (both can be found) (QML). User interfaces created with this technology do not have to be compiled but are instead loaded dynamically, which speeds up the process of adapting functionality. Since QtQuick has not been released until the then latest minor release of the API, which was Qt 4.7, this technology was not known to AviBit; therefore, the concerns about using it were big. Especially the idea of moving away from the well-known concept of QWidgets has raised lots of worries; a concept providing ready-made GUI components, that has been the base of Qt based GUIs for many years in the past. Although there were many concerns about using such a new technique, the first impressions, gathered through preliminary experiments, seemed very promising. These experiments consisted of studying the available examples from the Software Development Kit (SDK) and browsing all available online resources from the community, due to the lack of other available literature. For this reason we decided to extensively test this new framework and to do a lot of experimentation and prototyping to get a feeling for what it is capable of and to which extent it would limit AviBit's plans for the result of this work.

Initially we have identified the bridging of a QtQuick HMI and the application logic written in C++ as one of the major concerns, since a lot of the available examples solely deal with creating QML-only applications. While being a comprehensive showcase for the possibilities of this new technology, we had to figure out, how to create software that is not limited to simple mobile applications, like the photo viewers, clocks or games that these samples demo. It is our goal to create a full-fledged desktop application for the ATC business not some fancy animated, small-scale smartphone app, with not much logic behind it.

4.1.3 QtQuick Evaluation Details

The following paragraphs explain the basics of Qt's QtQuick module and at the same time explain, why we have finally agreed to use it as the basis for the HMI part of the system's client module.

During our extensive prototype phase we have tried to get an in-depth feeling for the technological possibilities and limitations of QtQuick and how to implement the required

features with this new technology. For all of these prototype experiments we have completely ignored AviBit's usual application design, which uses a lot of special code for different purposes like setting up an application, synchronizing time over the network, using shared memory to communicate between different processes, special logging mechanisms, network code, etc. We have decided that it is enough to get a basic application up and running, featuring what we thought is needed to get enough information on the framework, to be sure that it is ready for the real development of the required system.

The first goal was to find out how to extend QtQuick with custom components that would finally be needed because of its rather simplistic approach on offering ready-made components. Since it is not possible to implement items like wind roses with QtQuick on-board functionality, they have to be implemented in C++. This task turned out to be feasible using Qt's painting engine. This possibility also offers a very modular approach on implementing special widgets, as they can be used in QML like they were components natively provided by QtQuick. This has the big advantage that there is no need to touch the framework code later on. During this phase we have also tried to use the upcoming programmable shader support which is already in parts available in Qt 4.8, but will be an integrated part of Qt 5. With programmable shaders, which are described in detail in [Rost et al. 2009], it is possible to make more use of the power of modern day graphics cards. The use of special code, written in one of several available, specialized shading languages, allows to program the graphics card's shading units, to alter the processing of vertices, geometry and pixels. Shaders are not very much used in today's desktop user interfaces but once available they can be used to implement graphical user feedback in a multitude of different ways and create visually appealing notifications, feedback, etc. to the user, that otherwise would have to be processed by the main processor, instead of the better suited graphics processor. They can, for example, be used, to render effects like drop-shadows or highlighting of active items with a surrounding, pulsating glow. As we know from the gaming industry a lot of imaginative ways exist to use shaders. With support in GUI frameworks like QtQuick, these can be used for non-gaming applications as well.

The second goal in this phase of this project was to get an idea of how to combine a declarative user interface, created with QtQuick, with the power and flexibility of C++.

4.1.3.1 Custom Extensions to QtQuick

QtQuick's available set of graphical primitives is limited to drawing rectangles with optional rounded corners. Other geometric primitives are currently not supported. To test the implementation of custom drawn components, we took one important and more complex element, we would for sure need in the desired TIS, a so called wind rose or compass rose respectively and tried to realize it. From our experience with controllers we know that wind roses are a very intuitive way to get a quick reference of the actual wind direction. This information is very important, since, depending on wind directions and speeds the controller may be forced to change the active runway to another one, since too much crosswind, for example, meaning wind coming from the side, can lead to unsafe situations for landing and starting aircraft.

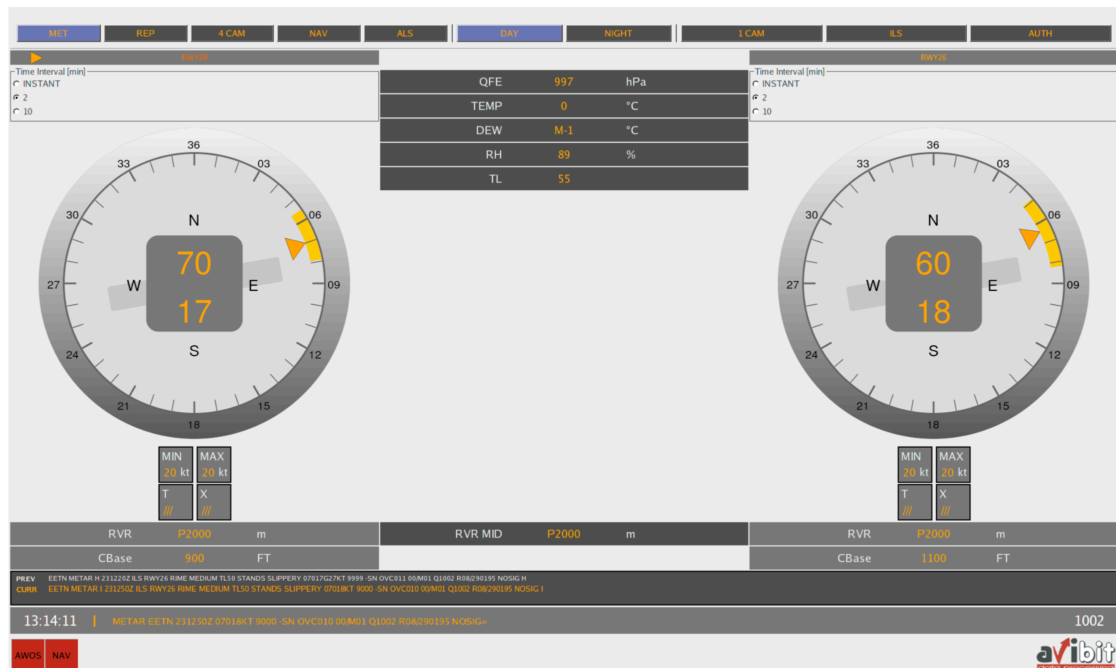


Figure 4.1: *A previous TIS created by AviBit. Next to different other meteorological values, placed at the top and the bottom of the display, it shows two windroses prominently placed to the left and the right of the screen. They show an arrow for the current wind direction and a segment of the circle for the average wind variation during the last minutes. In the center of the circles we see a schematic runway and, in the dark gray box, textual representations of the current wind direction in degrees and below, the current wind speed in knots.*

AviBit already had such a wind rose widget in place. The fact that this widget was written in Qt3 had the drawback that we could not use it for a quick integration into our prototype, due to the differences between Qt3 and Qt4. Investigating the drawing code of the existing wind rose, revealed how complicated it was to draw such an element with the techniques provided by the older Qt version. The fact that Qt3 did not support transparency on widgets, which means they always had a solid background except from complex workarounds with masks, forced the software engineer to write the drawing code, for all the text fields on the rose, presented in Figure 4.1, himself. It was clear that this would not be necessary anymore, when combining the drawing of the wind rose with simple texts created in QML. So we started implementing a first prototype of such a wind rose by inheriting from QtQuick's basic item class and drawing it with Qt4's Arthur Paint system (see [Digia 2013b]). After implementing such a custom item it is necessary to register it to the declarative engine used by QtQuick and it is instantly available in QML. All properties of such an item that need to be configurable in QML have to be made available through Qt's property system, which makes heavy use of Qt's Meta-Object system (see [Digia 2013c]). After the successful implementation of this task we started enhancing the component by adding configurable colors, which are also easily

added through QML's binding system, as well as animations to show smooth transitions when the wind direction changes. The drawing and usage of a custom graphical element was the first thing we have tried out and in letting us doing this with little effort, QtQuick proved to provide the necessary capabilities to implement custom components.

4.1.3.2 QtQuick HMI with C++ Back-End

The next step was to assess how QtQuick can act as the view, i.e., the GUI, for a data model fully implemented in C++. One of the general ideas we had in mind, when creating this framework was to move all data manipulation to the server part and have the GUI really only act as a view, at least as far as possible and meaningful. To test the feasibility of the API for this use case, we have implemented a data object, similar to the one planned for use in the real system. This simple prototype class was as a test class, locally simulating the server part and the reception of randomized data. This served to get a feeling for the interaction between C++ and the declarative engine. The data object used in this case was able to have different states, e.g., updated, invalid, etc. For the type of the data's value we used Qt's QVariant since we already had in mind to use this later on as a generic type for simple transmission of different data over the network. This idea was also implied by the fact that QVariant is one of the types that is supported by default in QML. The declarative engine also converts the variants automatically for primitive types, such as strings, integers, real numbers and even container types, such as lists and maps. Also very important for a generic use of QVariant is the fact that in contrast to Qt 3, Qt 4's variant type can be extended with arbitrary custom types. A quality that is very important when serializing data over the network.

The simple goal we wanted to achieve with this test code was, to get, e.g., text fields to show the actual value of the data's variant and to change colors, based on the current state of the data, as explained above. In fact this is again a very intuitive task, which can be fully implemented using Qt's meta object system and its properties. In addition QtQuick's state machine can be used to easily change colors if different conditions are evaluated. States are not limited to color changes, they can as well be used to change behaviors of mouse areas, text input fields and to change any other property that is accessible in QML. Another easy to use technique are transitions, which complement the above mentioned state machine by enabling the programmer to easily define transitions for state changes. It is, for example, possible to implement a simple color animation if a value turns from the normal state to the updated or invalid state and in any other direction respectively. These transitions can also be used to define arbitrary other animations like movements from one position to another, resizing, rotations, etc. or as well simple number animations that can be used to change numeric values in a continuous animation.

The basic interaction, i.e., data handling, between C++ and the declarative part in QtQuick is not only more than sufficient but opens up new possibilities for the visualization and behavior of the HMI, through the power of an integrated and easy to use state machine and transitions. GUIs in the past offered a very static experience but with QtQuick it is possible to create arbitrary user interface elements and the whole interface

can profit from well designed animations for, e.g., menus, pop-ups, transitions between different views and others.

4.1.3.3 Custom HMI Component Management

A tab like structuring of the GUI application would also be necessary for the proposed system and so we also tried to prototype this functionality. We did not want to take the easy but limiting way of having to include all pages that can be selected by tabs, in one master QML file. Instead, we opted to implement a mechanism, where we can define only the main window setup in one master file and move the page setup to a separate QML file. This file just has to be used in the master file to automatically load the required pages at application startup time. During this step, we learned how to load QML components from the C++ side and position them correctly in the scene and in the right declarative context to have access to the main window's functionality (e.g., JavaScript functions defined in the main window). How this works is, like many other possibilities of the new API, not very well documented. In the end, however, we have also managed to realize this functionality in our prototype through the creation of separate QML components.

4.1.3.4 Editor Capabilities

Another idea we initially had for the framework, was the inclusion of an editor, where the user should be able to choose from a list of predefined components and arrange them freely on a page of the GUI and save it afterwards. This functionality could be used to customize the product more intuitively then by writing QML files. The first step of this goal again proved to be quite easy by loading different components and drag them around, through the use of a mouse area, only enabled in a special editor mode. But for the first time during this prototype phase we found no feasible solution in QtQuick; none that enables the programmer to save the given scene back to a QML file, which would be needed to load the scene again. But since QtCreator, Qt's very own Integrated Development Environment (IDE) has a built in tool to design and edit QML components in a graphical way, we were sure that there has to be a way, we could do this too. But we were proven wrong. We had a deep look into the code of QtCreator's QML designer plugin and after an extensive investigation, of the scope of such an editor, we came to the conclusion that the effort needed to get something similar for our framework, would by far exceed the benefit of such functionality. Our estimate is that it would take multiple times the time to add the editor capability to the framework, than to develop the framework itself. For this reason we have decided that the editing features, at least more complex ones, would not be part of the design for the final framework, since it is also not an essential requirement, especially, when considering the flexibility and ease of use of the QML language itself.

4.1.3.5 Evaluation Summary

The outcome of the prototype phase, explained in this chapter, is our decision that QtQuick is the right choice to create the basis for a flexible and modern looking HMI framework, like the one AviBit asked us to create for their TIS. There have been discussions necessary to convince the company of the idea to leave the former QWidget approach and to move on to the relatively young but modern and powerful API that we have investigated. In the end we have achieved to agree on using QtQuick for the new system and could finally enter the design phase. QtQuick delivers every user interface related functionality that is required for the framework, as defined in Subsection 3.1.1 (Requirements and Goals), from colors and animations to components that are specialized for touch input devices, like gesture areas, that can be used instead of mouse areas.

4.2 System Design

In this section we present a design for the system that fulfills the requirements as discussed in Subsection 3.1.1 (Requirements and Goals) and that is based on the discussion in Section 4.1 (Technology Overview and Evaluation). For the design it is not sufficient to just focus on the HMI portion of the system; therefore, this section is split up into three major modules, a client module, a server module and a third module that copes with a generic way of letting interface processes communicate with the server.

4.2.1 Design Overview

This subsection shows a general overview of the whole system design as can be seen in Figure 4.2 and explains the general purpose of the different modules, leaving the deeper insights to their respective subsections. The parts of the system are the following:

- **Server:** In the center of Figure 4.2 the server module is shown. As the diagram illustrates, it is the heart of the system. The server has two main responsibilities, on the one side it shall handle all communication with external interfaces through dedicated processes and on the other side it shall provide the clients with all the information they need and, as well, handle communication coming from the clients to the server. It does process, store and distribute incoming data to all interested parties. The server is as well responsible to keep all relevant information in a persistent database to avoid losing any information in case the server has to be restarted. The operational server in its “exec” state also has to sync its own state to his twin, the “standby” server, which is waiting to take over operations in case of any failure of the “exec” server, whether it is a failing network interface, hard drive or the software itself. State, in this case, basically means, the same information that is also locally stored in the persistent file. As shown in Figure 4.2 the server is monitored by a Monitoring and Control System (M&C) to inform maintenance personnel, which has access to it if anything goes wrong.

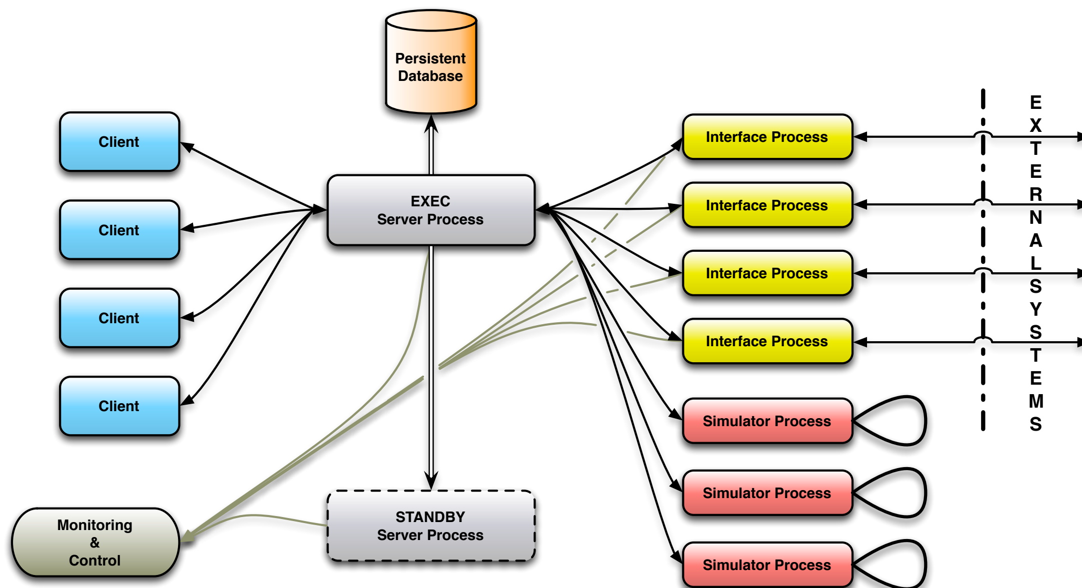


Figure 4.2: *This figure shows an overview of the system design. In the middle of the diagram we see one server process that is in exec mode, while the one below is waiting in standby mode. To be ready to take over operation, in the case of a failure of the exec server, it has to sync its contents to the standby process regularly. To the right we see the different interface processes that are connected to the active server, as well as to their respective external systems. The communication between the server and the interface processes works in the same way, as between the server and the simulation processes shown below. For the server it is not distinguishable, if it talks to a real interface or to a simulated one. On the left we see client processes that are connected to the server in order to retrieve data and display it. We also see that both servers, as well as the interface processes, are connected to a monitoring and control system that monitors system failures and triggers alerts and exec-standby switches, if necessary.*

- **Interfaces:** The next important part of the system are the interface processes. It is their duty to handle the communication to all the different external systems that are possibly connected to this system. These external systems may all have different ways to provide the information, our system is interested in. Some use serial connections for their communication, some use ethernet or any other way to communicate; some send the information on their own, some have to be polled to get data out of them; and nearly none of them uses the same protocol to transport information. So basically most of these interface processes, responsible to communicate with the external systems, implement a very specific, most of the time proprietary, protocol.

  After processing received data, the interface process shall forward all relevant information to the server, in a unified manner. So if we want to summarize their task in one word: they are translators. They receive information in a foreign lan-

guage and translate it to a common language understandable by our system and the other way around. At AviBit these interface processes are called Data Acquisition Processs (DAQs). This naming emerged from the fact that often times these processes only receive data, e.g., from radar or weather sensors but seldom send data on their own. So just to avoid confusion for the reader, we want to clarify that we call these processes DAQ, even if one of them actively sends data to an external system, like air field lighting control commands. Even if we describe the server as the heart of the system above, it is the existence of interface processes, which makes the whole system useful. It is the main goal of the system to provide an integrated view on many different external systems and this would not work without these DAQs.

- **Client:** The last major module is the one we focus on in this work, the client. The client is the user's window to the data received by the interfaces and stored in the server's database. It is the client's responsibility to provide an intuitive and informative way for the controller, to investigate what is going on in the different external systems, while at the same time it shall not reduce performance or impede safety. Ideally, however, it should increase both of the latter. The client in our design tells the server at startup time, what kind of data it wants to initially receive, so there is a separate connection maintained by the server with a separate state concerning each client and its data. It is also possible for the client, to change interests during runtime and request other data.

  The design foresees that the HMI part of the client itself is nearly completely separated from the rest of the logic. This assures the flexibility of the HMI concept. Data configuration and GUI layout can be freely intermixed depending on the use case without having to touch anything concerning the data module of the client. The details of this concept are further explained in Subsection 4.2.4 (Client Design).

4.2.2 Communication Design

After describing the main modules of the system and their major duties in the previous subsection, we now give a short overview of the most important concepts concerning the communication of these modules with each other, before discussing the separate modules in detail.

The client-server architecture presented in this system is not a broadcasting system, where every client receives all kind of data available from the server and then decides whether to keep the data or throw it away. Our concept relies on the use of individual data connections, where clients only get the information they want.

The data distribution and manipulation concept of the system is based on three different mechanics that function as follows:

- **Subscriptions:** This mechanism is used for automated data distribution by the server. The client can be configured to automatically receive updates on the subscribed data in real time or at certain delay intervals. The latter is used to avoid

too frequent updates, e.g., if a value received by a sensor gets updated in millisecond intervals, it may be enough to show updates on the HMI every two seconds, to reduce disturbance of the user by ever changing values. In this case the subscription would have a configured interval of two seconds and the server would wait for two seconds before sending any newer data. Independent of the subscriptions, the server stores all received updates, which is obvious since other users, i.e., clients, may want to receive updates in different intervals. Subscriptions are the most frequent used concept in the system, since most data shall be automatically received by the client. Subscriptions can be created and canceled dynamically to support, e.g., data visualization in temporarily visible elements. This feature also helps to minimize data traffic and keep client hardware resource consumption, used to process incoming data, to a minimum.

- **Requests:** Compared to subscriptions, requests work the other way around. Through the former, the server knows when to distribute data to connected clients, whereas the latter are used to tell the server to transmit data on the client's demand. This can be used, for example, in HMI components that should not be overwritten by data from updated subscriptions but where the users instead want to refresh a view by themselves. This mechanism removes automatism from the server, which on the other hand does not mean that the client may not implement an automatism to send requests.

- **Commands** The last communication mechanism does not work the same way as the other two. Whereas the former ones trigger data updates from the server to the client in one way or the other, commands are more of a universal technique to manipulate data on the client or even interface processes. In the simplest case a command can be used to directly update or replace a value stored on the server, with a value sent by the client. This is the client's way of changing data directly on the server. Commands can also be used to trigger more complex actions, like calculations that as a result, update different values in the server's database. Commands may as well be forwarded to DAQs, which are then responsible to process them. All of these actions can trigger a transmission of updated data to the client, if an appropriate subscription exists. Commands can also have parameters attached just like if they were local function calls.

The communication concept as it is designed, is the first step into a direction where every single piece of software can access a universal data aggregator, i.e., the server module of this concept, and use the data it needs or use the controls it offers. This is a very large step forward towards moving all different interface and data structure abstractions to the interface processes and making them transparently accessible through the server. Later on, the data processing and communication part of the client module can be easily implemented as a library instead of a standalone HMI, to just put the corresponding views, wherever needed, into other applications that need data from the interfaces connected to the server.

4.2.3 Data Item Design

To fully understand the underlying mechanisms of the system we also have to explain how data is stored and accessed. This does not mean the internals of the server module but the general concept of data items in the framework. Every data item managed by the server has a single unique identifier. This identifier is a text string that follows a simple dot notation to reach data items in a hierarchical manner. The notation is similar to what is often used in programming languages to access members of objects in an object tree and may look like the examples in Figure 4.3. The string before the first dot is used as a special domain indicator, which can be seen as the root of the data item's position. This domain indicator is used primarily by the server, to identify from which interface process a specific data item originates. For the client it has only informative value but, nevertheless, has to be known when configuring the client. The rest of the data item identifier can be freely chosen but most often it will contain strings, such as the 4-letter ICAO airport location indicator of a site or an identifier for a specific runway. It can also be used for special categories of the related interface, like, for example, a power supply unit from an airfield lighting system and its states may be located somewhere else in the hierarchy, than the states of the lights themselves, as shown in the example. The notation we use in the system is helpful in organizing the available data logically into groups of variables that belong together and makes it easy to find data items in the server's internal database. It is also used to access commands as they use the same naming scheme.

Concerning the type of a data item or variable we have decided to go for an approach, where the value is always stored as a variant type to support easier scripting on both, the server as well as the client side. Although this form of dynamic typing has to be accompanied by greater caution, to avoid errors, we have decided to accept this compromise as a trade-off for the flexibility this approach offers.

For the sake of simplicity we have also introduced a wildcard mechanism that is supported in subscriptions and requests, to be able to express the client's interest in whole groups of variables. The notation of this wildcard is also shown in Figure 4.3. If this technique is used, the server knows to send all available data items that are hierarchically at the same level of the wildcard or below.

Data items stored on the server have not only a value connected to them but also different sorts of additional information. It is possible to set a *valid from* and a *valid to* timestamp, which is useful if, for example, a value is expected to be updated in certain intervals. The client can then use this information to indicate an outdated value to the user, which on the one side avoids that the controller works with invalid values and on the other side indicates that there might be a problem with the sensor or the whole interface. The values can also have a history if needed for a certain use case. The number of history entries can be limited by a maximum age or a maximum number of history entries. All data items can as well have scripts attached, to be automatically triggered if an update for the value is received. This feature is very handy to trigger calculations, e.g., if the current wind value for a sensor changes and the server receives an update, the script can be used to calculate the average wind speed for a certain amount of time

```
met.loww.qnh
afl.loww.psu.status
afl.loww.rwylights.status
afl.loww.rwylights.lampsfailedpercentage
afl.loww.*
```

Figure 4.3: *Examples of the Data Item's Dot Notation*

in the past, from the value's history entries and the new value and automatically update another variable on the server. This scripting functionality is something we consider to be one of the most powerful features of the design, since it allows for a lot of flexibility without having to touch the core framework, except for extending the overall scripting capabilities.

An additional important detail of the data item design is not directly related to the storage of the values and their retrieval through the naming scheme introduced above. It covers the data retrieval by the client, through the communication mechanisms discussed in the previous subsection. Since the client can subscribe or request the same data multiple times with different intentions, the server as well as the client shall be able to differentiate between these intentions. For this reason, in all communication between both processes that is related to data distribution, a separate requester name has to be sent. The requester is an arbitrary string that enables the client to identify the source of certain subscriptions or requests, which may be different components of the HMI. The mechanism also enables the server to deliver all data to the correct addressee without having to know any other specialties in the client's behavior.

4.2.4 Client Design

In this subsection we treat the design of the client module, the most important module in the focus of this work. As the client module shall allow for easy adaptation, the design presented in Figure 4.4 shows that there is a distinct separation between two main modules. On the one side there is the core module of the client, which is responsible to handle the basic tasks. This core module consists of the following submodules:

- **Networking:** The network module of the client application is responsible for all tasks related to communication with the server. This includes the handling of the network connection itself, like establishing, closing and maintaining the connection through a bi-directional heartbeat sending mechanism, to be able to easily tell on both sides, if the connection is lost. This module is also responsible to send subscriptions, requests and commands to the server, as well as to receive the corresponding responses and most important, the values from the server's database. All low-level network related functionality is fully encapsulated by this module, so the rest of the application does not have to bother with it.

- **Data Storage:** This module is responsible to store all received data values from

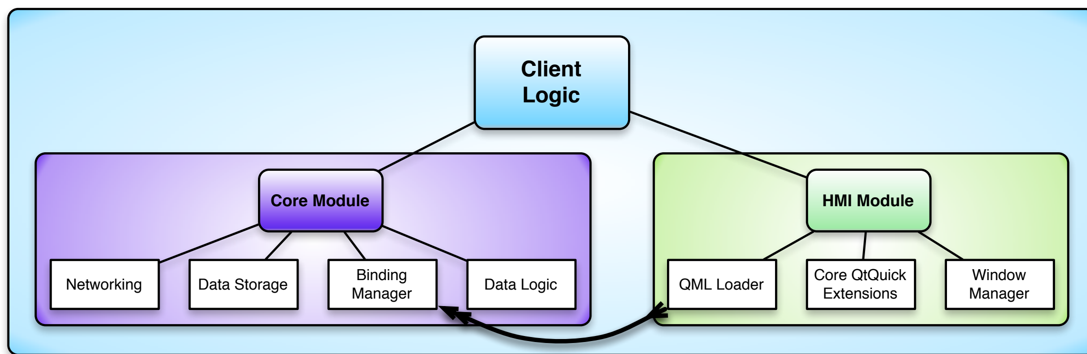


Figure 4.4: *This figure shows an overview of the client design. We can see that the client, globally managed be the client logic module, is split into two almost independent submodules, as illustrated by the two separate boxes. To the left we see the core module, with its several components that are responsible for networking and data handling. In the right box, in contrast, we see the part of the system that is responsible for any HMI related functionality. The only relevant connection point, between these two modules, is illustrated by the thick black arrow at the bottom. It points from the QML loader module to the binding manager, because all data related configuration happens in QtQuick and is, once loaded through the former module, implicitly passed to the latter, which then takes care of the bindings.*

the server. It is the client side version of the server's database. Depending on the use case of the created application it can contain all variables of the server or a subset of it, since it only contains data for variables that are subscribed or requested by the client. The data storage's function is, to act as a central point, where all data is stored, however, it does not process any variable in a certain way. Client side data items are responsible themselves to know when they expire and have to handle their validity according to the information received by the server. Direct access to this database from the HMI part is not intended in this design, instead every access to a data item has to be encapsulated by a binding explained in Subsection 4.3.2 (Client Details).

- **Binding Manager:** This part of the client's core framework is responsible to handle all bindings available. Since the term "binding" is newly introduced, we have to explain what we mean with it in the context of this work. We only use it in the client's design, since it handles data differently than the server. On the server each data item only has one representation or instance, whereas the client can use one and the same data item from the server in different representations. The most basic explanation of a binding is that it binds to a name, the name of a certain variable stored in the server's database in this case. The name of the binding is supplemented by a requester, as described in Subsection 4.2.3 (Data Item Design). There are four different kinds of binding related mechanisms available in the client's design. Three of them are the client-side representations of the mechanisms explained in Subsection 4.2.2 (Communication Design) and are called

subscriptions, requests and commands respectively but they are only responsible to communicate with the server. The fourth type, however, that we simply call binding, is responsible for what data is actually used on the client. So even if the client subscribes to all variables via a wildcard, only the ones, where a specific binding exists, are stored. The rest will be discarded.

The binding manager also connects data storage entries directly to their bindings, so they can be updated directly, without involving the binding manager to search for relevant bindings if a data entry gets updated.

- **Data Logic:** The data logic module is responsible to take care of the bindings and their needs to communicate with the server. It is responsible to inform the server about available subscriptions, as well as command and request invocations, via the networking module. It also processes responses from the server. This is the module's responsibility related to the outside world. Internally it is responsible to forward data updates from the server to the data storage.

The second main module is responsible for all HMI related functionality. This module consists of the following important parts:

- **QML Loader:** Most of the actual HMI can be implemented in QtQuick and the only thing needed to support this is a possibility to load QML files, which is the core functionality of this module. QML file loading is done in two steps. At first, for every binding category described above, a separate QML configuration file can be created, to aggregate all bindings that are needed during the whole lifetime of the application in a central file. The bindings loaded through these files are then made available in the HMI itself, which is loaded in the second step. This step basically consists of the loading of a main QML file that includes the whole declaration of the HMI.
- **Core QtQuick Extensions:** This module extends the functionality of the application beyond the very basic possibilities that are available out of the box in QtQuick. It is not so much a closed module as it is a set of components we have decided might be useful in such an application, even if in specific client applications, it may not be wanted, to use them at all. On the one hand there are basic GUI elements, such as scrollbars, image views, document views, directory views, etc. and on the other hand these components include specific items to create a tab interface and provide navigation for these tabs, as this is as well nothing that is natively supported by QtQuick. Grouping specific information that belongs together on tabs, is one of the concepts we plan to support in the system, since it allows, for example, to create separate pages for airfield lighting maps, document browsing, different reports and others, apart from one main page that aggregates the most important information. This module is the one we intend to expand mainly over the life time of the framework, to include more and more reusable components to enhance the user experience and have standard GUI components available, with a common look and feel.

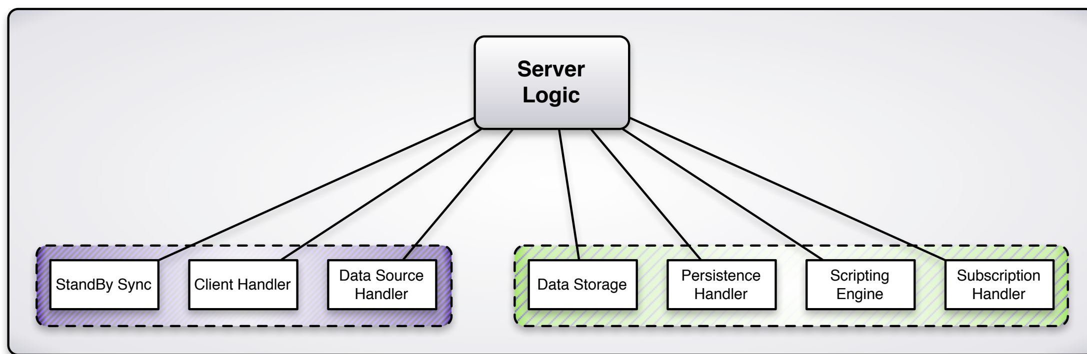


Figure 4.5: *This figure shows an overview of the server design. We can see the different sub-modules, grouped together functionally, to network related modules to the left and data related modules to the right.*

Adding new functionality to this module shall not result in the need to touch the rest of the framework most of the time and it shall be basically the only module that needs to be touched if new possibilities should be made available HMI wise.

- **Window Manager:** Since we don't want to limit the application, to only have one main window, the design also offers the possibility to detach or clone tabs, as well as specific components, into separate windows. Therefore, the core HMI module includes a simple window management module that is responsible to create new windows and show the correct content in the resulting child windows. When a window is closed the window management module is responsible to move detached parts back to their original position in the main window's scene, or to delete cloned components.

All of these modules are managed by a main client logic that is responsible for the correct setup of all of them and their relations with each other. It sets up the main window, triggers the loading of the necessary QML files, initiates the client's connection to the server and connects the right bits and pieces for the system to work. During the runtime of the system, however, it has not much to do.

4.2.5 Server Design

The server, as the heart of the system, as we called if before, has a lot of duties. To fulfill all of these, it needs different capabilities, which are again split up in modules. These modules that are shown in the diagram in Figure 4.5 are described in this subsection to provide a high-level overview of the server's design.

- **StandBy Sync:** For security reasons, in case of a failure of the server process or its hardware, there has to be a backup server process running. This module is responsible that the standby server always has the same state as the operational

server. Therefore, it has to sync the whole database of the exec server to the backup process, whenever data changes.

- **Client Handler:** This part of the server is responsible to handle connections to clients. It maintains relevant information related to the clients and is responsible to assure that messages get dispatched to the right addressees. The module on the one side forwards any incoming client query to the relevant server module and on the other side, passes any outgoing messages to the client. Each incoming message from the client gets flagged with the client's unique identification string and passed on. This assures that all other server modules do not have to take care about clients.

- **Data Source Handler:** The data source handler, in contrast to the client handler, is responsible to maintain connections to the interface processes. It handles incoming data from the DAQs and signals that variables have changed, so they can be changed in the data storage accordingly. It is also used to forward commands to the interface processes, for example, to change the state of a certain lighting system status.

- **Data Storage:** This module encapsulates all access to the server's database and provides the necessary functionality to store and retrieve all relevant information about data items, like history data, validity periods, as well as obviously the values of the data items themselves. It is also responsible to signal to the subscription handler, that a variable has changed, so it can trigger its updates to the clients.

  The data storage is not restricted to data from external systems. It is also possible to configure variables managed by the server itself or the client, so, in an extreme use case, the module could also be used, to store the currently selected tab from a client but most often this will be used, to store states where no external interface is available. A prominent use case, for this kind of server-only variables, is the content of message boxes to share information between clients.

  The module is also responsible to trigger the execution of scripts that are attached to variables. This is useful to automatically update other values in the database, like the calculated average wind speed. It could as well be used to automatically trigger an ATIS update that is sent to an external ATIS system through an interface process, afterwards. To support this kind of processing, the data storage module has to maintain all kinds of scripts that are relevant for these behaviors and it has to know, which variables are dependent on other variables, to correctly trigger the relevant scripts.

  The module is also responsible to perform data maintenance tasks, like removing history entries from data values that are no longer needed.

- **Persistence Handler:**

  All data entries in the data storage have to be kept persistent, so the process can be safely restarted, without losing any data previously received from interfaces.

This module is responsible to serialize the whole database to the persistence file on the disk.

- **Scripting Engine:** The scripting engine delivers the functionality that is used by the data processing module to execute scripts. It is responsible to load scripts and provides the scripts with the necessary access to data storage entries, as well as the possibility to write entries to the database. The module is also responsible for the execution of scripts that are scheduled to automatically be executed, whenever a data entry is updated.

- **Subscription Handler:** The subscription handler is responsible for client subscriptions. This is necessary since subscriptions require different handling mechanisms than requests or commands, which only require one time processing. If the client sends new subscriptions, it stores them and, subsequently, triggers the sending of necessary updates to the clients if changes in the data storage occur. It is also responsible to handle intervals as described in Subsection 4.2.2 (Communication Design) and remove subscriptions once they get canceled.

- **Server Logic:** The server logic in the end is the part that glues everything together. At startup it loads the server's configuration, sets up the data storage and reads possible data from the persistence file into the data storage. It also establishes the connections to the configured interface processes through the data source handling module. It sets up all necessary connections, so that incoming commands, requests and subscriptions get dispatched to the responsible modules.

4.2.6 Interface Process Design

The data acquisition processes, which are probably the most essential pieces to make a system, like the one we describe in this work, functional, are also the ones we cannot specify completely. For this reason the design in Figure 4.6 can only show, how such an interface process looks like in general.

Since the DAQs always have to implement a very custom protocol we cannot design the exact internals of these processes, neither for the protocol they are using nor for the internal workings and data processing. Therefore, the design of the interface processes basically is reduced to the communication with the server. Whatever these processes do internally, the communication with the server shall be the same. For this reason we explain the basic principle of the communication workflow between the server and an interface process. At system startup time the server connects to all configured DAQs. Then each of them registers the identifiers and other information of all data items the process will deliver to the server. This design avoids that these items have to be configured separately at the server, as long as the server does not do additional processing with the data. During operation the process receives updates from the external systems, processes them if necessary and forwards them to the server in the format already described. If there is bidirectional communication available the process can also receive commands from the server, which have to be processed and forwarded to the external interface and

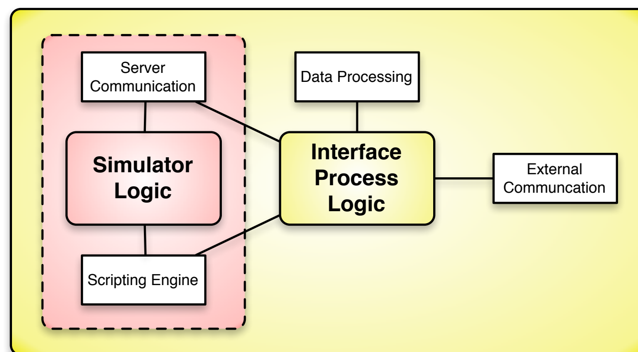


Figure 4.6: *This figure shows an overview of the basic design of the interface and simulator processes. As can be seen, the simulator process, illustrated on the left side, is very similar in design to the interface process, outlined to the right. The latter one only needs to be extended with a module, to implement the protocol of the external interface and another one, to do the data processing. Since the interface process also has access to the scripting engine, the latter module is not strictly mandatory, if the processing can be done through scripting as well.*

return a potentially available result of the command to the server. Simple processing can also be done through the scripting module that is available to the interface process instead of implementing it in the process itself.

As can be seen in Figure 4.6 we have designed and implemented a simulation process, following the same scheme that the interface process design proposes. This process is a powerful tool, used to mimic an interface process during the development, where real interface processes with correct input data are not available. It provides very flexible simulation capabilities, again, through the use of a scripting module.

The complete behavior of this simulation process can be created by configuration and scripts, to recreate the behavior of the final system, even if bi-directional communication is needed. This helps greatly in testing server and client behavior, before the real interface process is implemented.

4.3 Implementation Details

In this section we cover implementation details of the framework's separate modules.

4.3.1 Overall Details

Before we start to delve into further details of the separate modules, we deliver a short summary of the technological concepts that are used throughout the system.

The whole framework is based on the Qt framework, that is already explained in Subsection 4.1.1. This spans all parts of the implementation, from the use of Qt's networking capabilities to the use of its container classes. All scripts that are used on the server side and for the simulator process as well, are written in JavaScript and

are loaded, parsed and processed through Qt's QtScript module. On the client side JavaScript is used as well but through the scripting functionality provided by QtQuick itself, which offers the use of this well established scripting language directly in QML code.

Messages that are sent over the network use a proprietary messaging system developed by AviBit to serialize and deserialize data. Since this messaging system also has the capability to serialize messages to files, it is used for the persistent database as well, instead of, e.g., an SQL based approach.

One very important detail about the implementation is that it makes heavy use of Qt's signals and slots concep (see [Digia 2013a])t, which is a very handy way to implement event driven programming. Most of the different modules described in Section 4.2 (System Design) are connected via this mechanism and communicate with each other in this way. The most important of these connections are discussed in the following subsections.

In the following, we discuss the most important details of the three different processes. The main focus is put on the internals of the client process but we also explain important implementation details of the other two processes, next to class diagrams, illustrating the relevant classes of all three of them.

4.3.2 Client Details

For the implementation of the client, several details are important to know. First of all, the GUI related code that is written in C++ and hardwired to the rest of the client's framework is kept to a bare minimum. The only parts that fall into this category are the ones that are responsible for window management. The only C++ related GUI classes created are those that are used, to implement certain components that are not available in QML and also cannot be recreated through the combination of available components. These components, however, are in no regard tightly coupled to the client code, since they can be loaded at will, through their use in QML code. The only thing needed to make this work is their availability through a plugin. For this reason, they are neither covered in the class diagram, shown in Figure 4.7, nor explained in the following discussion of the main workflows of the client:

- **IMBindingBase**

  IMBindingBase is the base class for all binding related classes explained later on. It offers the ability to set variable and requester names, as explained in Subsection 4.2.3 (Data Item Design). More important, it also emits a signal any time one of these values changes. This update signal can also be used by inheriting classes, to signal updates on their own.

- **IMBindingTemplateBase**

  This template class serves as an intermediate class between IMBindingBase and the specialized bindings. For each instantiation it is responsible to register the created object to IMBindingManager. This is done automatically at construction

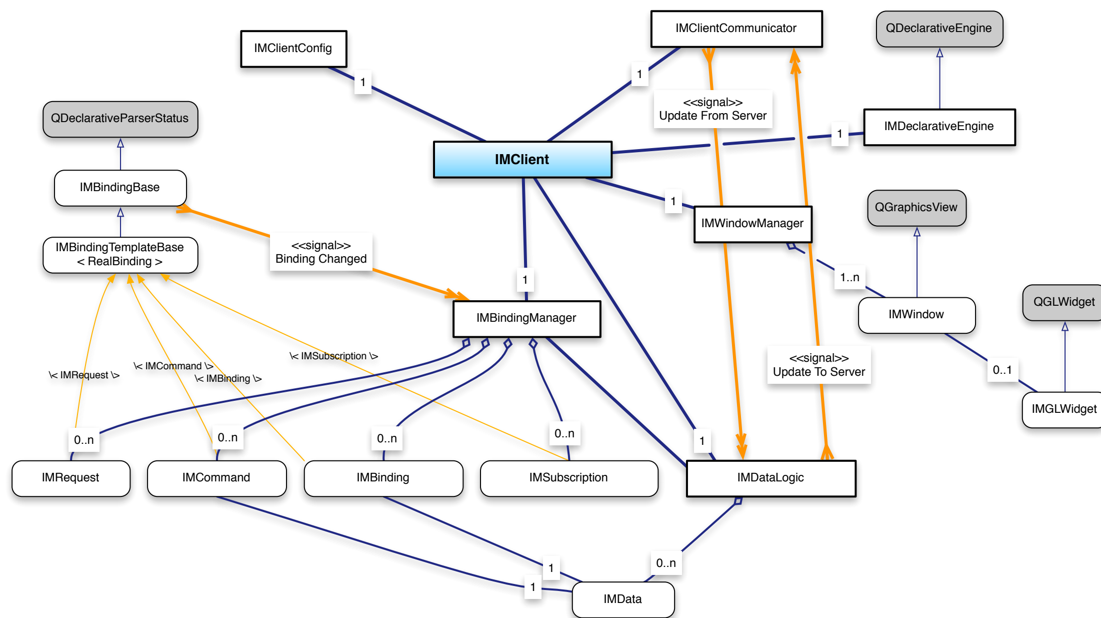


Figure 4.7: *This is the class diagram of the client process. All entities shown in white, with rounded corners, illustrate classes that can have multiple instances, whereas the others are limited to one instance, managed by the main IMClient instance, highlighted in the middle of the diagram. In the gray boxes we present the base classes that are interesting to see, related to the creation of an HMI with QtQuick. The thick orange arrows represent the most important signal-slot connections used, to make the client function, whereas the thin yellow ones illustrate template instantiations.*

time and, at the time of destruction, the object is automatically deregistered from the binding manager, to avoid dangling references.

- **IMSubscription**

  The IMSubscription class is used to communicate to the server that the client is interested in certain data items. This is also a handy way to use the wildcard mechanism explained in Subsection 4.2.3 (Data Item Design). Subscriptions are sent to the server at construction time or any time the variable, requester or subscription interval changes. At destruction time, the subscription is automatically canceled from the server.

- **IMRequest**

  This class can be used to manually trigger data updates through QtQuick and, therefore, offers a method to invoke it from QML code.

- **IMCommand**

  The IMCommand class is a very important one, since it offers the functionality to invoke commands on the server or one of the interface processes. Like the

IMRequest class, it offers an invocation method but in contrast to the former, it can also handle parameters that are passed through to the server. Sometimes it is also needed to lock a variable on the server, e.g., if someone is already changing the contents of a message box. For this reason commands, like IMBinding objects, also reference the related IMData object, to know the lock status of the server variable. This allows, for example, to block certain HMI components from receiving input.

- **IMBinding**

  This class can be considered to be one of the foundations of the client, since it servers two different, very important purposes. On the one side, it is responsible on the client side, to know, which data is really needed and which incoming data can be discarded. On the other side it is the single point of entry for the HMI to the data storage. Only IMBinding objects can be used, to access a data item's value, its validity and when it was last updated.

  For each object of this class a reference counter is increased for the given variable and requester in the data storage, which is managed by IMDataLogic. It can be configured with an interval that indicates how long an update should be considered as recent, which can be used in QML for special color coding or animations. Signals are triggered automatically if the time is over. The same applies for the validity state of the binding, which it derives from the IMData object it references. It also has an auto-subscription capability, so no separate subscription has to be configured.

- **IMData**

  Objects of IMData are used to store the data received from the server. They store the value and data type of the related variable, the time when they received the last update and the locking status. If any of these values changes, they emit a special signal that can be used by IMBinding and IMCommand objects to act accordingly. Since multiple of the latter two classes can reference the same IMData object, all of them are reference counted, to avoid the need to store multiple instances of the same variable.

- **IMBindingManager**

  The binding manager references *all* created objects that are derived from IMBindingTemplateBase. All registered bindings are connected to IMBindingManager's handler methods via the signal that is emitted, if a binding changes. The binding manager is also responsible to connect the individual signals from the bindings to the correct slots of IMDataLogic, to which it has access. For example, commands have a signal that is emitted, when the command should be triggered. This signal is connected by the binding manager, to the relevant slot of the data logic, which then knows that a command has to be sent to the server, if its invocation signal is emitted.

  This class is also responsible to connect the signals for updates on IMData objects

to IMBinding and IMCommand objects, so they recognize when the status of data, received from the server, changes.

The binding manager is not only responsible for the initial connections but also has to maintain them throughout the lifetime of IMBindingTemplateBase derived objects. So, whenever such an object changes or gets deleted, this class has to reconnect signals if needed or do the necessary cleanup operations, i.e., disconnect obsolete signals and slots.

- **IMDataLogic**

  IMDataLogic's single instance is responsible to dispatch all outgoing signals, received from the bindings, to the IMClientCommunicator instance and, in the other direction, to store every incoming data update that is not discarded, in its container of IMData objects, which in return, trigger updates to the relevant bindings.

  It is responsible to send new, updated or canceled subscriptions, as well as invoked commands and requests to the server. To avoid the sending of, e.g., multiple commands, it also maintains a list of pending requests to the server and removes them upon receiving a corresponding response from the server.

- **IMClientCommunicator**

  This class encapsulates the low-level networking activity and provides IMDataLogic with a convenient interface, so it does not have to bother with the details of the communication, at a network interface level.

- **IMDeclarativeEngine**

  This class is derived from QDeclarativeFramework and is the essential class needed to work with QtQuick. It is responsible to load and parse QML component declarations and lets us instantiate these components to receive usable GUI objects.

- **IMWindowManager**

  This class is responsible to create the main window and to open, close and handle child windows by moving the above-mentioned GUI objects to the correct windows and their respective scenes.

- **IMWindow**

  The windows in the client application are derived from QGraphicsView. QtQuick also provides a QDeclarativeView class that delivers a convenient form, to get a view with the correct settings for QtQuick and a QDeclarativeEngine at once but we selected this approach, since with the current API, it is not possible to share the same QDeclarativeEngine between different views. This would have caused us troubles when using multiple windows, functionality and also memory wise, since one QDeclarativeEngine uses a large amount of memory. To have the option to use Qt's OpenGL renderer, each window can also have a QGLWidget as its viewport, as can be seen in the class diagram in Figure 4.7.

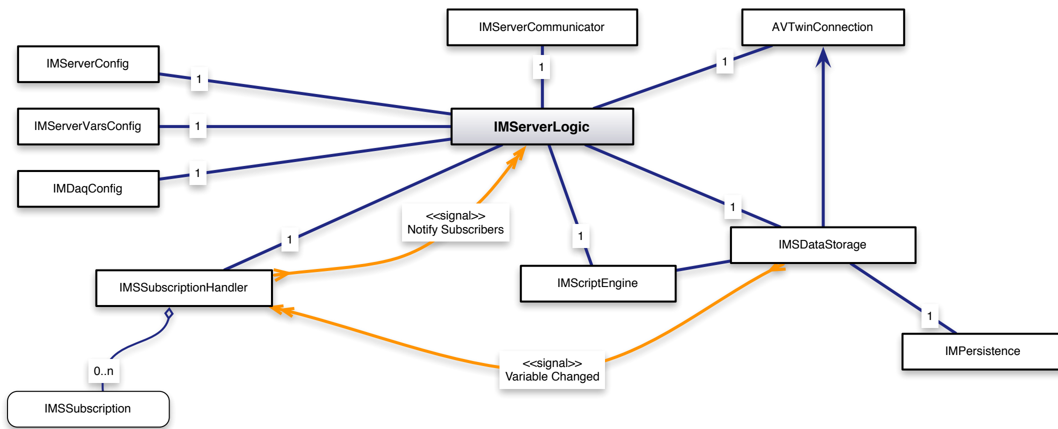


Figure 4.8: *This is a diagram of the server process' main classes. It shows the most important classes and the most important signal-slot connections, needed for the server to work. The latter ones are represented by orange arrows.*

- **IMClient**

  This is the class that is responsible for the application's general setup. It loads the configuration through an IMClientConfiguration instance and initializes the IMBindingManager, IMDataLogic, IMWindowManager, IMClientCommunicator and IMDeclarativeEngine objects, connects all relevant signals and slots between the communicator and the data logic and triggers the creation of the GUI through the loading of the QML files by the declarative engine, which are then passed, as a scene, to the main window.

To summarize the basic interaction between the core module and the HMI, all IMBindingBase derived classes, mentioned in the list above, can be configured, extended and even scripted purely in QML code and at the time of their object construction, they are automatically registered in the core framework's binding manager, which does all the rest of the maintenance during their lifetime. On the QML side, which is used to create the HMI itself, all updates to these bindings, are then automatically triggered through related signals and slots provided by Qt's property system.

4.3.3 Server Details

Similar to the client, on the server side only the really framework specific logic is implemented in C++. Every functionality that is related to airport specific situations can be configured and scripted. This offers great flexibility and again, as is the goal for the whole system, reduces the need to touch C++ code, if the system has to be adapted to a new site. We have even developed a basic script library that offers the most needed calculations through simple JavaScript function calls that can be triggered from the scripts that are attached to variables.

In general, only the variables that are not received by the interface processes, have to be configured separately on the server, apart from the cases where the aforementioned variable specific scripts are used because these scripts have to be correlated to their respective variables.

In the following list we discuss the internals of the server's main classes:

- **IMSDataStorage**

  The IMSDataStorage is the responsible to manage all data related tasks. It keeps the configuration for each variable, applies updates, maintains the histories of variables. The module also knows which scripts are dependent on which variables, so the correct scripts are processed through the script engine, if a recalculation is necessary. There is also a rollback mechanism integrated into the data storage. If an update with multiple data items contains corrupt data from the sensors this mechanism can be used to restore the sane state of the database.

  Another very important task of IMDataStorage is to signal to IMSubscription-Handler that a variable has changed, which makes the whole subscription concept work.

  When variables are updated, the data storage also triggers updates to the persistent file and the standby server.

- **IMPersistence**

  The IMPersistence class is used by the data storage to keep the persistence file of the database up to date whenever data changes.

- **AVTwinConnection**

  This class has two purposes. On the one side it checks the availability of the exec server's standby counterpart, or the other way around and on the other side, it syncs the current database state to the standby server by sending delta messages on each data update. The receiving server then applies this delta to his database and his persistence file, so both servers have the same state.

- **IMSSubscription**

  Objects of this class are used to handle client subscriptions. They are identified by the client id, as well as the requester and variable names. They maintain the timer for the intervals of a requested subscription and when they have triggered an update notification for the last time. They know if the client also needs the history of variables, when receiving an update and, if yes, the maximum age or maximum number of history entries.

- **IMSSubscriptionHandler**

  The IMSSubscriptionHandler is responsible to maintain all IMSubscription objects created for the different client instances. For each incoming data update, it gets notified by the data storage that a variable has changed. It then queries its database

of subscriptions for suitable candidates and if they need to be updated. If candidates are found and they need an update because the minimum update interval has timed out, a signal with the updated variable is send to the server logic which further processes it. If the interval has not timed out, a timer is started and the update to the client is postponed, until the timer finishes. This assures that the client always receives the last valid value of a variable, even if a minimum interval is set.

- **IMScriptEngine**

  This class provides the server with its powerful scripting capabilities. It is responsible to load scripts and execute them, when triggered by the data storage. It also loads available JavaScript libraries and provides them to the scripts to extend their possibilities. It is also responsible to cancel scripts that do not respond and log errors, for easier debugging.

- **IMServerCommunicator**

  This class is responsible to manage all communication of the server. On the one side it listens on open ports for clients to connect, assigns unique identification strings to them and dispatches incoming requests, commands and subscriptions to the server logic and forwards messages to them. On the other side, it establishes connections to the configured interface processes and handles incoming and outgoing messages. So, in contrast to the clients which connect to the server, DAQs are acting as servers themselves and the server acts as a client. This is needed, since it can be required that the interface process is needed to serve multiple servers.

- **IMServerLogic**

  The server logic class is responsible to handle the overall application flow. At startup time it loads a multitude of configurations through instances of the following classes:

  - IMServerConfig: This configuration class is responsible to deliver the overall configuration of the server, like the settings for the exec-standby connections, or general port settings.
  - IMDaqConfig: This class manages hostnames, ports and domain names for all configured DAQs.
  - IMServerVarsConfig: This is the most complex of the three configuration classes. It handles settings for all variables that have to be managed by the server itself. It stores, whether separate values for each client shall be kept for variables, how long variables can be locked, whether they are virtual or not, which means only available by script evaluation, and other information.

  During operation the server logic is responsible to forward incoming data to the data storage and send outgoing data to clients. It also forwards commands to interface processes and dispatches the responses accordingly. In short, it is the overall message dispatching central.

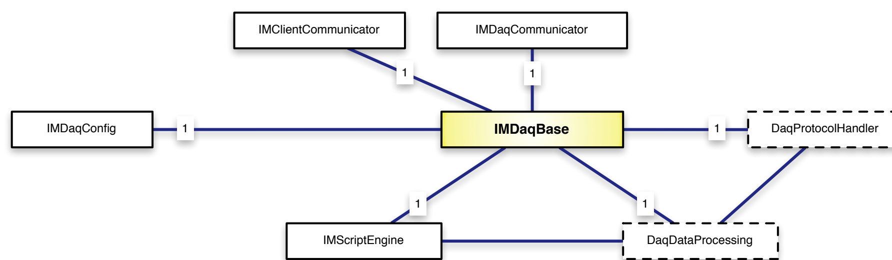


Figure 4.9: *This is a class diagram of the simulator process, which at the same time builds the base for an interface process. The boxes surrounded by dashed lines illustrate the classes that are needed, if an interface process shall be created.*

4.3.4 Interface Process & Simulator Details

In the following we explain the main classes of the simulator process we have created, which is basically a subset of an interface process that only needs to be extended with the necessary classes, to realize external communication and specific data processing. The main classes are:

- **IMDaqCommunicator**

  The class provides signals for incoming commands and provides the functionality to send data to the server. This makes the module easy to be used by future DAQs as they do not have to take care about maintaining connections and other low-level details related to the communication between server and interface.

  In the server details, we explain that in general, the variables delivered by interfaces, do not have to be configured on the server. For this reason the interface process communicates a list of the variables and commands it offers to the server, so the server knows, which variables will be available to the clients and how to dispatch incoming client commands. This behavior frees the process of adapting the server to a project, from redundant configuration tasks and is done through this class.

- **IMClientCommunicator**

  This class is the same class that is described in Subsection 4.3.2 (Client Details). Through the use of this class the interface process is enabled to act as a client. This functionality can be used if the process itself, needs data from the server, to do its processing. With this construct we avoid the need for the server to handle DAQs and clients separately, while providing the interface processes with all the same possibilities of subscriptions, requests and commands that each client has.

- **IMScriptEngine**

  Again, this is a reused class, this time from the server's implementation. The only difference in here is that it loads different scripting libraries, depending on the task of the interface or simulation that shall be handled.

- **IMDaqBase**

  This is the base class that handles the management of the simulator process. It loads the configuration of the variables and the DAQ itself, through an instance of IMDaqConfiguration and sets up the connection classes and the script engine.

  To implement an interface process this class just has to be derived and extended with the processing classes for protocols and data.

Chapter 5

Results

After designing the framework we have implemented prototypes for three different customers of AviBit that will all be using systems based on the proposed framework in the future. Two of these prototypes are rather similar and, therefore, make it easy to analyze the framework's adaptation capabilities for similar setups. Contrarily, the third setup required a completely different HMI, so we had the possibility to see how flexible the architecture of the framework is, when it comes to different usage scenarios. In the following chapter we present the results of the implementation of the framework, the different prototypes and a summary of customer experiences with these prototypes.

5.1 Evaluation Scheme

The main goal we have set for this project is to create a flexible framework for interactive client-server based information systems, one that supports rapid prototyping cycles and short shipping times while offering the possibilities, to create modern graphical user interfaces. In the forthcoming parts of this results chapter, we evaluate the different parts of this goal for their success. In the remainder of this section we explain the process of this evaluation.

Since we have developed three different prototypes, we discuss and analyze each of them separately, starting with a short analysis of the core framework beforehand.

From a commercial point of view, the most important success factor of this framework are the development efforts needed, to adapt the system to the requirements of a new project. For this reason, we have tracked every hour that was spent for development, starting with the first lines of the core framework, until the finalization of the three prototypes as they stand now. We discuss these efforts during the remainder of this chapter. We begin with an introduction about the general method of tracking and an explanation about the interpretation of the numbers gained. For every prototype we present, we also include a short summary about the client specific efforts, since our focus for this work is the HMI part of the system. However, to make a qualified statement about the efficiency of the framework, it is equally important to know the rest of the efforts needed to adapt the system. For this reason we also analyze the general efforts in an additional section to summarize our overall observations of the framework's capabilities. As a last efforts evaluation step, this section also includes a comparison of the project adaptation times between one project from the past, the project that initially

highlighted the urgent need for a redesigned application concept, due to its unacceptable project adaptation times and its flawed design.

The second part of the formulated goal is not related to efforts but to the user interface capabilities offered by the framework. This is difficult to measure in numbers; therefore, we present figures for all of the three developed prototypes to present what we have developed and to show to the reader, what is possible with the framework. In addition to the figures, we explain several HMI features in detail, to deliver backgrounds on our considerations. During this project, we took part in workshops with controllers to present the prototypes to them. So, as a counterpart to our own impressions about the framework, we also present opinions from controllers, the future users of the system. These workshops tend to be rather impulsive and controllers formulate their thoughts in an unstructured manner, when criticism or ideas come to their mind. When discussing the results, we bring the comments, we have gathered, in a more structured form to present their experiences with the presented prototypes and what they imply for our framework. Since the presentation of the prototypes has been embedded in AviBit's schedules with customers, the setup for the workshops was different every time. For this reason we also outline the setup of the workshops within the parts of this chapter where, we present each prototype.

At the end of this chapter we include a summarizing discussion to wrap up and interpret the results presented before and to discuss details of the results that did not fit in the sections before.

5.2 Efforts Analysis Preface

In this section, we present the overall efforts, we have tracked for the implementation of the core framework and the prototypes we have realized. We explain how we figured out these numbers and what trade-offs, we have made to keep the effort of tracking itself, at a reasonable level. This section serves as a general explanation of the presented numbers, whereas the detailed analysis of the efforts is covered in the following sections, when the core framework and the prototype results are discussed in more detail.

Table 5.1 shows the overall efforts spent during this project. Tracking of working times for this project was done in hours but for the purpose of this work all efforts used in later comparisons are converted and rounded to whole man-days, since this kind of granularity makes more sense and is significant enough to compare tendencies in efforts. What can be seen in this overall view is, that we have split the tracking of times in two parts, the client module on the one side and a combined tracking of the efforts, needed for the server, the basic interface process and the simulation, on the other side. We made this decision in the beginning, since our main interest is in the client's performance and to avoid the generation of too much overhead through the tracking of too many different tasks. The same reason of keeping tracking efforts low, also applies for the client times, which are combined values for the HMI module and the core module. In the numbers derived from the results in Table 5.1, which are presented in other tables, we have assigned relative percentages for the separate parts of the combined values, based on our

	Client		Server & Interface & Sim		Overall Efforts
	hours	man-days	hours	man-days	man-days
Core Framework	1031	134	942	122	**256**
Saudi Arabia	320	42	238	31	**72**
Parchim	153	20	204	26	**46**
Austria	182	24	224	29	**53**

Table 5.1: *This table shows the overall efforts spent during this project. The initial values in hours are converted to man-days where a day has 7.7 hours. The first line contains the efforts spent for the core framework, whereas, the lines below summarize the efforts spent on the several prototypes. To keep the efforts for tracking times at a reasonable amount, the tracked times are split up between client efforts and the combined efforts for the server module, the basic interface process implementation and the simulation environment.*

experience. For these tables, we also take into account a certain percentage of overhead that is generated by tasks that are not directly accountable as efforts in the sense of this work, where we focus on implementation, especially for the prototype implementations.

For the combined server, simulation and interface efforts, in general, we know from practice that for the core framework, the server part is responsible for most of the efforts spent (see Table 5.2) in this area, whereas for the prototypes, it is the simulation environment that accounts for the majority of hours (see Table 5.3). For the client part, the efforts were rather similar for both modules in the core framework implementation phase but during the prototype phase the HMI adaptation efforts significantly outweigh the core module. These facts are reflected in the weighted breakdowns in percent, shown in both of these tables.

Finally we have to add a note concerning the accuracy of the separation between core framework and prototype efforts, shown in the tables of this chapter. One of the difficulties during the realization of the prototypes was, that we had to develop the prototypes in parallel to the core framework, since the workshops and presentations to showcase the demos, have been planned according to schedules, not influenced by us, but by project plans agreed with the customers. This obviously makes it a lot harder to accurately track or differentiate the exact times invested in the prototypes, in comparison to the hours spent for the implementation of the underlying design itself, as outlined in Section 4.2 (System Design). On the one hand, it is not easy to always separate working hours between such, spent for the demo and those, spent for the core system and on the other hand it is difficult to assign times to one or the other, if a certain feature of the core system, that is needed for the prototype, is not yet implemented. Therefore, we want to point out that the times, measured and used for the analysis of adaptation times for the prototypes, are not hundred percent accurate but since these errors are not of a significant dimension and are distributed in both directions, we count the tracked times as a sufficient approximation for the adaptation capabilities of the framework in different situations.

	Client		Server & Interface & Simulation			Overhead	Total
	HMI Module	Core Module	Server	Interface	Sim		
Core Framework	60 %	40 %	65 %	15 %	20 %	10 %	256
	121		110			26	
	72	48	72	17	22		

Table 5.2: *This table shows the efforts spent in man-days to implement the core framework. The calculated efforts are taken from Table 5.1, reduced by a certain amount of assumed overhead and split up proportionally, to account for the functionality, they have been spent on. These numbers do not only include implementation times but also time spent for technology evaluation, system design, etc. Efforts spent in meetings or coordination tasks are counted as overhead.*

5.3 Core Framework

Before we describe the results of the prototypes, we discuss the implementation of the core framework itself.

Table 5.2 shows the efforts spent on the core framework. These numbers basically include the core framework as outlined in Section 4.2 (System Design), where the majority of the functionality is ready, with the exception of special server mechanics needed in an operational system, like exec-standby switch functionality and data synchronization capabilities between servers.

On the client side, most of the efforts accounted to the HMI module, include the multitude of components created in QtQuick that fit into the "Core QtQuick Extensions" mentioned in Section 4.2.4 (Client Design). These components include everything, from the most primitive GUI elements like buttons, scrollbars and contextual menus, to more complex ones, like prototype versions of directory browsers, message boxes, image viewers, Portable Document Format (PDF) viewers and others. In addition, we have developed different ready made animation components that can be reused in combination with other components, like fading animations, to show or hide elements, color changing animations, usable, for example, to indicate validity changes or updates to bindings and others. Additionally, we have created a rudimentary online color editor that has already proved to be a valuable addition during the creation of the prototypes. This tool can be seen in Figure 5.4.

As Table 5.2 illustrates, the efforts for the client part are slightly higher than for the other parts together, with overall efforts of around 230 man-days for the design and implementation including prototyping for technology evaluation.

5.4 Prototype Results

In this section we present the prototypes in the chronological order, we have started to work on them. This does not necessarily mean that we have finished them at the same time, but obviously, components needed in more than one prototype, need more time the first time they are created. Every prototype is described in the necessary detail,

	Client		Server & Interface & Simulation			Overhead	Total
	HMI Module	Core Module	Server	Interface	Sim		Client & Server
Saudi Arabia	90 %	10 %	20 %	0 %	80 %	5 %	
	39		29			4	72
	36	4	6	0	23		45
Parchim	95 %	5 %	20 %	0 %	80 %	5 %	
	19		25			2	46
	18	1	5	0	20		24
Austria	85 %	15 %	30 %	0 %	70 %	5 %	
	22		28			3	53
	19	3	8	0	19		31

Table 5.3: *This table shows the efforts spent in man-days for the different prototypes realized, during this project. The calculated efforts are taken from Table 5.1, reduced by a certain amount of assumed overhead and split up proportionally to account for the functionality, they have been spent on. Efforts considered as overhead are requirements clarifications, communication with customers and similar tasks that are not directly accountable as adaptation times in the sense of working on the prototype. Totals in blue boxes are grand totals, whereas, orange totals only calculate client and server efforts.*

focusing on features not available for prototypes that are explained earlier. We work out differences, to explain the adaptations needed for each. We split the three prototypes into two "system types", since it makes more sense to discuss two of them together because of their similarity and the third one separately because of its big functional differences.

We discuss all prototypes in a consistent manner, following a scheme were we introduce the prototype, explain the characteristics, differences or novelties in the HMI, discuss the efforts as shown in Table 5.3 and summarize the controller's comments, critics and ideas that we collected during the workshops, where we presented the adapted system.

5.4.1 System Type 1 - The Typical System

As the headline already tells, we consider the two prototypes presented in the following paragraphs, to be the typical system. The reason for this is that, as already outlined in the first chapters of this work, the goal of this project is to create a framework that is well suited to integrate other systems into one homogeneous system. This is exactly what the following two systems do. Both of them are systems where the customer wants to remove clutter from the controller's working position and replace it with a system that combines all former systems into one HMI, extended by several additional automated features not available before. It is the "typical system", since this is the kind of system that until now, most customers requested.

The general design of the two presented systems is similar and reflects what we mean by "typical system". Both of them feature a main page that aggregates most of the information that is needed to be aware of the current situation. This includes information

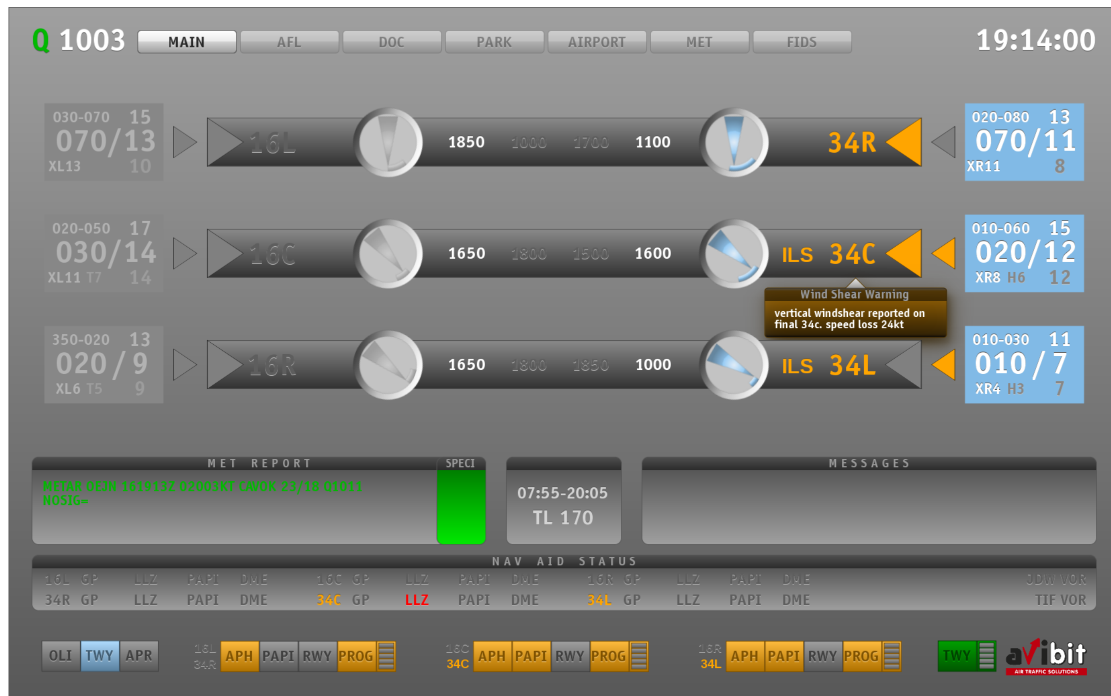
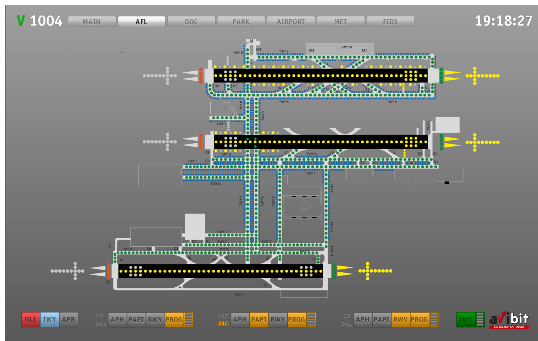


Figure 5.1: ***MAIN page:*** *This is the main page for the Saudi Arabian system, containing aggregated information from different systems. There is wind information available for three different runways. The arrows, next to each runway, indicate, wether the runway is active for arrivals or departures and according to this information, wind information is highlighted, like the boxes to the right or grayed out, like the boxes to the left. At the bottom we see an airfield lighting monitoring bar. Above this bar the operational status of the navigational aids is shown. Above we see actual met reports to the left and also a message box to the right, to let the controllers share specific information. In the middle there is additional information.*

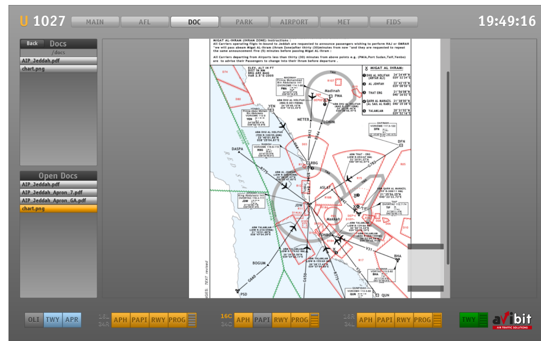
like wind and weather sensor data, weather reports, ATIS code, airfield lighting information, runway in use, the status of the navigational aids and others. Then, there is a dedicated airfield lighting page available, as well as document viewing pages and pages to retrieve different ATC related reports. In the following two subsections, we describe the differences and specialties of the two prototypes.

5.4.1.1 Version 1 - Saudi Arabia

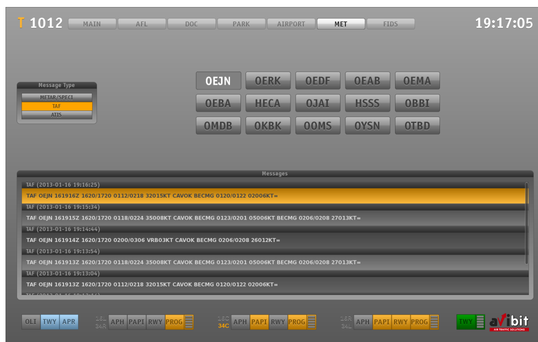
The first prototype created is targeted to be deployed on five different airports in Saudi Arabia. Its creation was split up in two phases due to the fact that the system was shown in two different workshops, one of them happening in a very early phase of this project. For the initial workshop, the prototype only included the functionality for one site, whereas the second version contained the configuration for all five sites.



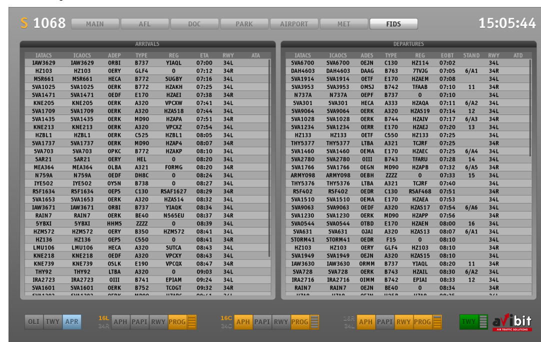
(a) ***AFL page:*** *On this page, the user can see additional information about the airfield lighting system, in addition to the bar at the bottom which is available on all pages, to get a more detailed overview of the system.*



(b) ***DOC page:*** *On this page, the user can view documents like PDF files and images. These documents can be seleceted from the file browser in the top left corner of the page. Open files are shown in the list in the bottom left corner and can be closed from there.*



(c) ***MET page:*** *On this page, controllers can select from a list of different weather report types for the airports represented by the buttons. The reports in chronological order are shown in the bottom half of the page.*



(d) ***FIDS page:*** *This page shows additional information about arriving and departing flights from the airport's FIDS.*

Figure 5.2: *These are the other pages for the the Saudi Arabian prototype. The PARK and AIRPORT pages visible in the tab bar are left out here since they are essentially the same as the DOC page, described in Figure 5.2b, with a special selection of documents. These pages differ from site to site but the differences are only related to the contents they present and not conceptionally.*

Saudi Arabian Prototype: The HMI

The major parts of the HMI components used in this prototype, had to be created for the first time for this prototype. This includes the main page layout elements, such as the bar that shows the schematic runway, the wind roses, the message boxes, airfield lighting bar at the bottom, document viewers, file browsers and most other elements that can be seen in Figures 5.1 and 5.2.

An unusual specialty of the Saudi Arabian project, is the already mentioned deployment to five different sites. Even though the available systems provide different interfaces on all five sites we wanted to keep the main page layout similar for all of them. We set this goal to keep consistency and lower the efforts needed during workshops to agree on designs. The framework, we have developed, has fully fulfilled our need for flexibility in this case, since we have managed to develop a layout, where the different configurations for the sites that are shown in Figure 5.3 only need minor changes in configuration files, to respect the different available runways, navigational aids, airfield lighting setups, etc. The different interfaces at the sites, however, can be fully encapsulated by the interface processes.

Saudi Arabian Prototype: Implementation Efforts

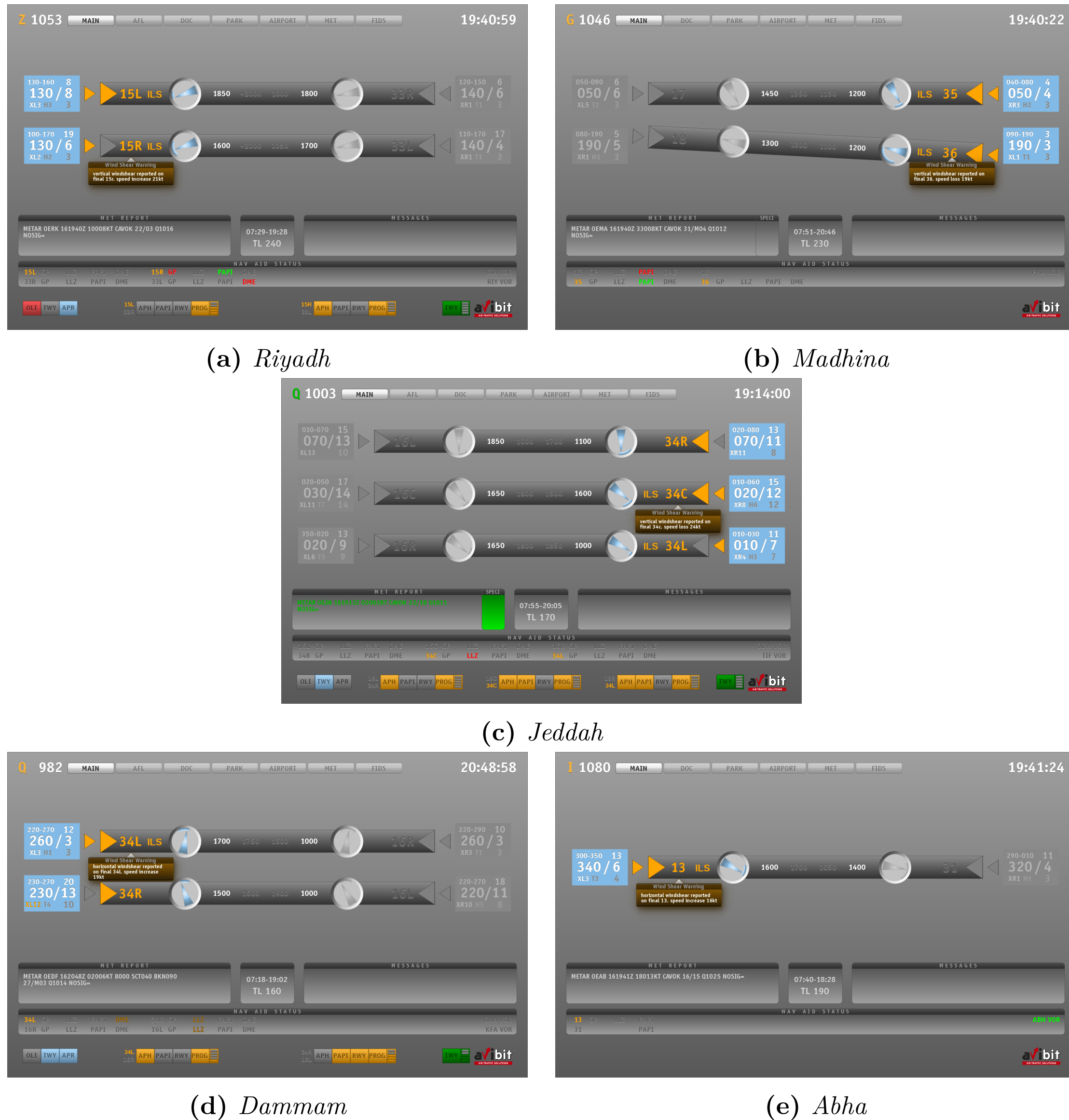
In Table 5.3, we see that the efforts to implement the client's adaptations to the HMI, took less than 40 man-days. Facing the fact that the adaptations had to be done for five different sites, this is a fairly reasonable amount of time. Even if this doesn't mean that one site would have needed only a fifth of the time, this is, nevertheless, a good value, especially when considering that this is the first prototype created with the new framework.

Saudi Arabian Prototype: Controller's Comments

We attended two different workshops with Saudi Arabian controllers. In the first, one we presented a very basic prototype, since at the time the whole framework was in a very early stage of development. For the second workshop, both, the framework as well as the prototype, have been in a near final stadium. During both workshops, we talked to multiple controllers from each of the five different sites. Both times we started with a presentation and a live demo of the system and afterwards let the controllers experiment with the system themselves.

The first version of the prototype did not have the possibility to detach tabs. This was strongly criticized by the controllers as they fear to miss important information or updates while, for example, browsing through a document on a different page. The feature was planned for the system anyway but this confirmed our assumption that such a feature may be requested by controllers. For the second workshop we have implemented this feature and received generally good comments.

Another initial complaint of the controllers was related to the color of the windshear warning popup shown in Figure 5.5. This popup was implemented due to the controller's request during the first workshop, to include windshear information on the main page, instead of the initially planned separate page. While implementing this component, we

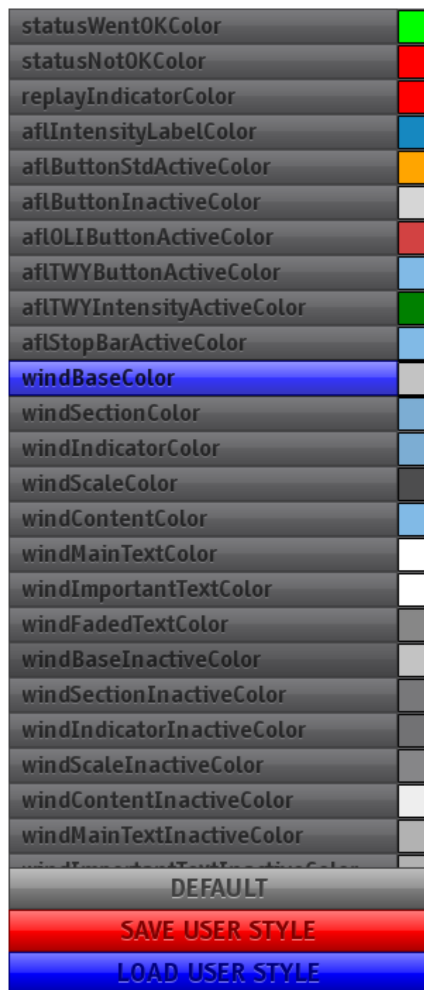


(a) *Riyadh* (b) *Madhina*

(c) *Jeddah*

(d) *Dammam* (e) *Abha*

Figure 5.3: *Comparison between the main page layouts of the system for the different Saudi Arabian airports. The sites feature different runway layouts, including wind information, have different sets of navigational aids and differ in airfield lighting monitoring availability. Figure 5.3b even shows a layout where the runways are not parallel, which is also reflected in the system.*



(a) *Color Selection Window*

(b) *Different Color Schemes*

Figure 5.4: *This is an example of the Saudi Arabian prototype with different color configurations, configured via the online color configuration dialog shown in Figure 5.4a*

Figure 5.5: *A windshear pop-up that fades in, when windshear conditions are serious and fades out if the wind conditions improve.*

initially colored it red, based on our understanding that windshear is something very dangerous. We have then been discussing this with an Austrian controller, who told us that aircraft are able to land during windshear conditions without troubles, as long as they know the conditions and that this is not necessarily an alerting situation. So we decided, to not color it red and instead gave it a more neutral color. When presenting it to the controllers in Saudi Arabia the great majority of them, when first viewing the popup, asked if we could change its color to red. After some discussion during the workshop, also including the Austrian controller mentioned above, we settled to color it yellow for the final system, to indicate a warning. This is an interesting example of how controllers with different backgrounds, may it be education or experience with such situations, tend to classify things differently.

From our experience during this project, these workshops in general tend to develop a momentum, where controllers start to argue about design decisions amongst themselves and interestingly, most of the time, after discussing, they just come up with a joint decision. For the Flight Information Display System (FIDS) page (see Figure 5.2d) of the prototype, for example, they wanted the order of the columns to be changed and as described, after some internal talks, they found a solution, which seems to fit for all of them. Technically with the proposed framework and the developed prototype this is basically no effort and if requested we could have changed this in a matter of minutes during the workshop itself. From a developer's point of view, however, this saved us from a lot of efforts to figure out a good and, above all, acceptable design by ourselves.

Another joint request was to add a search bar to the weather reports page shown in Figure 5.2c, to be able to search reports for airports that are not available in the preconfigured list of sites. It is again not much of a technical effort to implement this but takes a lot of fear from the controllers about not having access to important information.

Otherwise the prototype has received positive reviews and proved to us that the general design of the system is acceptable by the controllers.

5.4.1.2 Version 2 - Parchim Airport, Germany

The following prototype is suited for the needs of an airport located in Germany. It is similar in concept to the systems created for the Saudi Arabian airports discussed above but differs in functionality due to a different set of available systems that shall be integrated.

Parchim Airport Prototype: The HMI

The HMI created for this prototype basically reuses the layout paradigm introduced with the Saudi Arabian client but as can be seen in Figure 5.6, the main page here differs significantly from the ones shown in Figure 5.3. These different layouts are a good example, to show that even if the systems are similar in purpose, they will almost never be identical, as long as airports do have such a variety of different systems, from different vendors in place.

Apart from the different main page layout of this system, we even had to integrate a page including a simple webview to show the contents of a specific webpage. Another

special feature that we had to implement for this prototype is a user interface concept to retrieve information about restricted areas of the airspace. To both, the controllers and to us, it was not entirely clear, how we should design this feature, which resulted in the two concepts compared in Figure 5.9 and the implementation of both of them. The circumstance that lead to this uncertainty is one, that is from our experience, often inherent in such projects. At the beginning, it is not always clear, how the interface to the external system will look like. In this case, for example, at the time the prototype was presented for a major project milestone, it was not even clear if the integration of these restricted areas will even be possible. In the end it may be dismissed completely; therefore, it is important to keep implementation efforts of prototype features low.

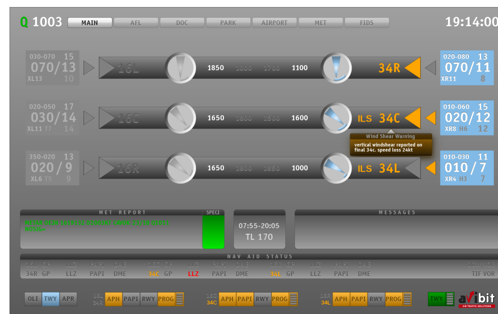
One of the additional requirements for this system is that the airfield lighting control bar, shown at the bottom of Figure 5.6, shall be integrated into another system. The design of the client module supports the easy realization of this request. Since we can load any declarative user interface we want to, it is easily achievable, to create a separate client application, that loads only the airfield lighting component from the full fledged application. This can be seen in Figure 5.7, where the reduced application is just placed above the main electronic strip application on the screen. For the user, the system in operation looks like one integrated system, since both applications have no window decorations. This is a much simpler and efficient approach, than to directly integrate these controls into the other application and it can be achieved, with little efforts on the development side.

On the main page of this prototype, there shall be a button to select a value from a list of different possibilities. This button, however, shall not disturb the user during the default, passive usage of the system, as an information screen. Therefore, we implemented a button that fades in smoothly when the mouse is hovering above it, to indicate that there is a clickable element. After the selection is made it smoothly fades out and only leaves the label on screen. A similar behavior is also used in the third prototype described in this work. Implementing functionalities like this, has also turned out to be very straight forward using the QtQuick framework, with its easy to use animations and state machine mechanisms. A sample sequence of the behavior is shown in Figure 5.8, although the printed version misses the smooth fading transitions between the steps.

Parchim Airport Prototype: Implementation Efforts

As we can see in Table 5.3, the client side efforts for this prototype are significantly lower, than those for the Saudi Arabian prototype, implemented earlier. This prototype needed only half of the time of the first one. We identify two main reasons for this: The increasing technology expertise and framework experience are one reason and the optimizations in the implemented components, the other one; components of the core framework that we had implemented for the first prototype but improved in the meantime.

Another experience with the realization of this prototype is the implementation of the two concepts for the restricted flight areas, described in the HMI part. As mentioned above, it is important, especially for uncertain features, to keep the implementation or adaptation efforts on the low side. For this feature, the effort to implement both



(a) *Saudi Arabian main page layout for direct comparison.*

Figure 5.6: *This is the main page for the system developed for Parchim Airport. The layout differs from the Saudi Arabian main page, presented in Figure 5.6a due to the different integrated systems.*

Figure 5.7: *Integration of the airfield lighting control bar, shown at the bottom of Figure 5.6, as a separate application, on the screen of another system. Aside from the graphical differences in the user interface components, the removal of window decorations suggests one integrated system, instead of two separate ones.*

of the concepts, including a full simulation, was only a matter of a few hours, which leaves us confident about the flexibility of the system. No core functionality, or line of C++ code respectively, had to be written for these concepts to be implemented. This fact holds true for most of the adaptations done for the prototypes. The main part is the creation of declarative HMI code in QML files and scripting for the server and simulation environments, without the need to recompile. This grants customers the chance to try out and experiment with fully interactive demos of the real system, which would otherwise be too expensive to implement for features that are not fully specified.

Parchim Airport Prototype: Controller's Comments

The Parchim system was shown to controllers during a test run of the system, where they had to accept the functionality of the HMI. The controllers expressed mainly minor issues with the user interface. They requested, for example, other intervals for the time, an updated value is displayed in a different color or different font colors. They also wanted the document handling to be more user friendly but this functionality was in an

(a) (b) (c) (d)

Figure 5.8: *These images show a sequence of the button behavior implemented for the Parchim Airport system, where a button shall only be visible if necessary, i.e., when interaction is needed.*
5.8a: *In the beginning the button shows only the label.*
5.8b: *After moving the mouse over the button, it fades in, to indicate that there is something clickable below and to provide feedback about the click itself.*
5.8c: *A menu with selectable items fades in, when the button is clicked. The button itself is visible as long as the menu is visible as well, even if the mouse does not hover over the button anymore.*
5.8d: *After a selection is made and the menu closes, the button turns invisible again, leaving only the label with the new selection behind.*

early stage at the time.

One major complaint was, again, the possible miss of information from the main page or the special info page of their system, when another page is active. Instead of detaching windows, this time, the controllers requested another behavior to overcome this issue. To be exact, they formulated two slightly different solutions to the problem. Both solutions require the tab buttons on the top of the window to be highlighted if a value is updated on either the main page or the info page. However, on the main page it shall stay highlighted as long as the value is considered to be new or until the controller switches back to this page. For the info page in contrast, which consists of only message boxes, only switching to the page shall end the highlighting of the tab, without any automatic timeout mechanism. The implementation of these requests can be fulfilled with minimum efforts by extending the QtQuick based HMI slightly but the request shows that a sophisticated solution is needed, to prevent the controllers from missing information.

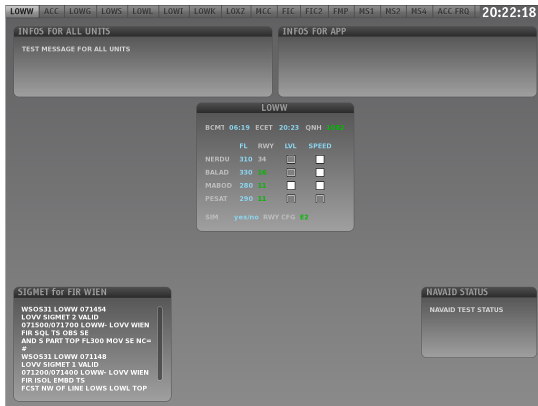
5.4.2 System Type 2 - The Special Purpose System

In contrast to the former two explained systems, the one presented below is significantly different. The purpose of this system, which is created for Austrian ATC, is to replace an older system with a new one that offers similar functionality but a modernized user interface and additional features. We do consider this system to be a custom system, since it does not contain most of the special features of the typical systems explained above. There is no airfield lighting integration, or direct integration of weather systems, no document viewing functionality or navigational aids status information, automatically derived from hardware sensors. The tool created, is a supporting one that helps tower, approach and center controllers from Vienna International Airport, the different Local Approach Units (LAUs) and even military units to keep in touch, know each other's

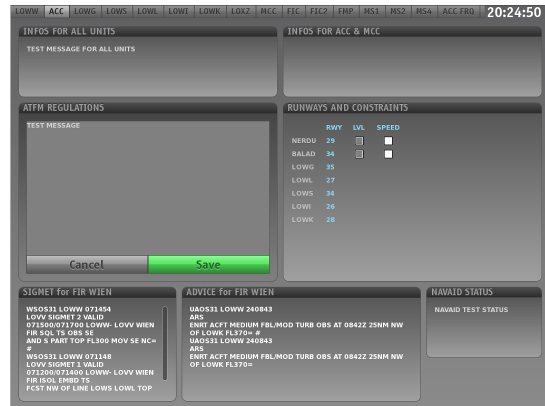
(a) *First concept of the NOTAM page.*

(b) *Second concept of the NOTAM page.*

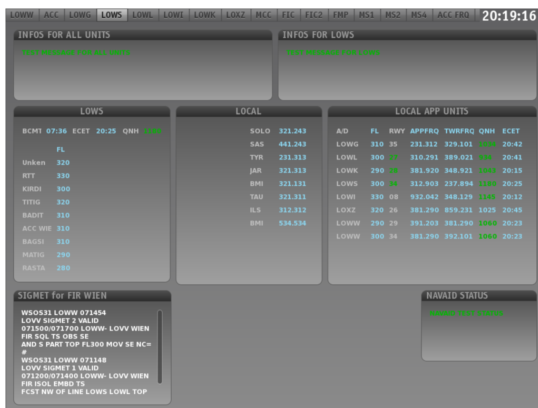
Figure 5.9: *Two different concepts for a specific page in the Parchim prototype to retrieve or present information about restricted flight areas. The user interface idea, in the upper half of the pages, looks completely different in both concepts.*



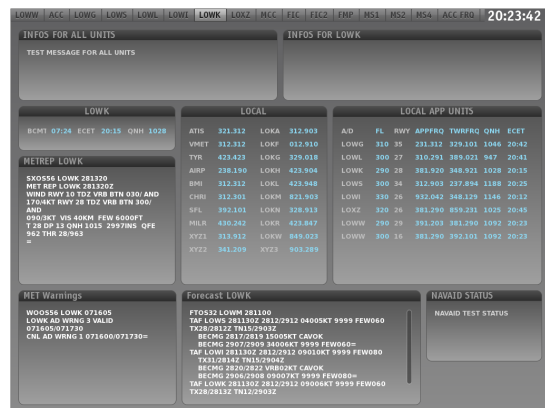
(a) *Page for Vienna Airport.*



(b) *Page for area control center with message box in use.*



(c) *Page for Salzburg Airport.*



(d) *Page for Klagenfurt Airport.*

Figure 5.10: *A selection of special local pages developed for the Austrian prototype. Most of these pages have an individual layout since they have a need for special information that is not needed by others.*

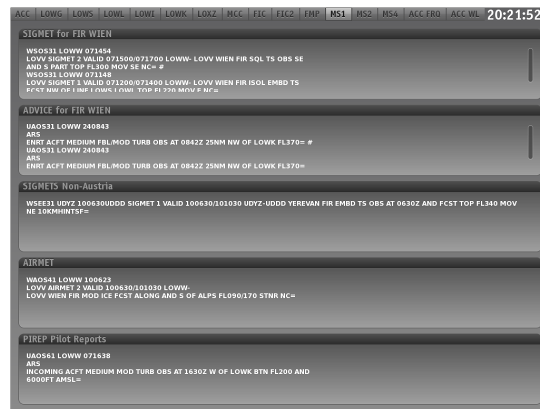
radio frequencies and look up different reports. It also contains other features, described in the following paragraphs.

Austrian Prototype: The HMI

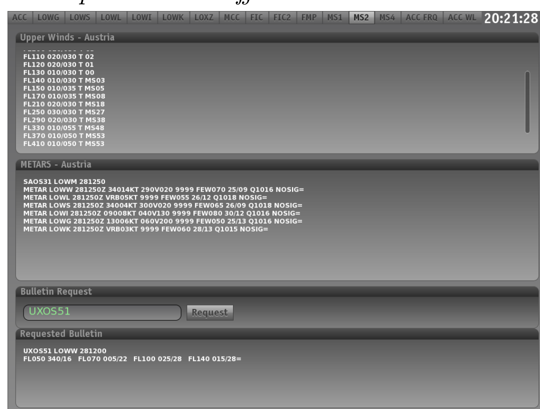
For the HMI of this system we have implemented a lot of custom components and improved existing ones. As can be seen in Figure 5.10 and Figure 5.11 this system features many pages (around 20), most of them with a different layout and special functionality. We do not explain each operational functionality, since this it is not the goal of this thesis but we show examples to further explain the possibilities of the framework and what we have achieved using the power of QtQuick.

In Figure 5.13 we show an improved version of the system developed for prototype number two (see Figure 5.8), where interactive elements are only visible if interaction is needed. This mechanism serves two purposes in this system. On the one hand, the

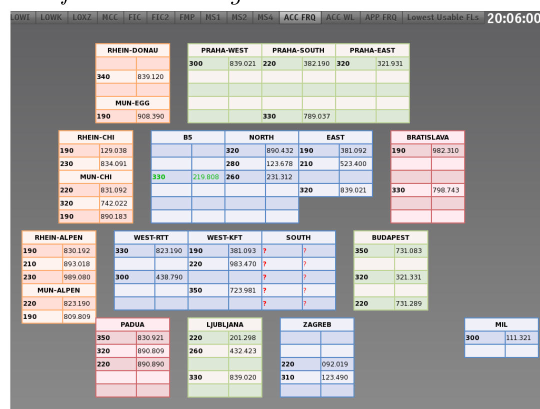
(a) *A page for special snow and volcanic ash reports where different sites can be selected.*



(b) *An entirely passive page which shows different incoming bulletins.*



(c) *Another page that shows bulletins but with the possibility to request any bulletin from the server via a search box.*



(d) *A page where radio frequencies for different regions are arranged and different frequency sets, also called phases, for each region can be edited.*

Figure 5.11: *An example set from the special pages of the Austrian prototype that are shared between different sites.*

(a) *Interactive elements are visible.*

(b) *After clicking on one of the fields, a custom-made dialog opens to change the entry. The* Save *button is disabled as long as the entered value is not valid. To know which field is edited, the dialog points to the still visible button of the selected field.*

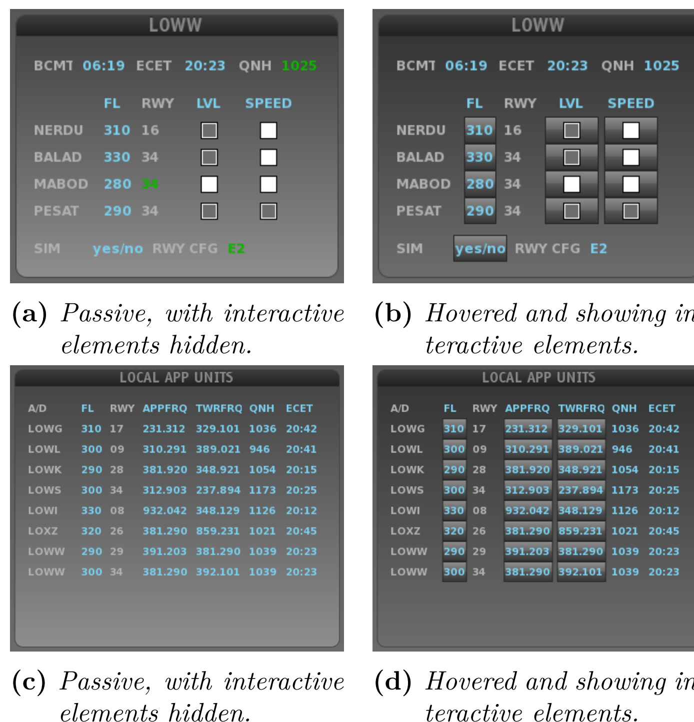
(c) *The* Save *button is enabled, once the entry gets valid.*

(d) *After saving, the field is updated and displayed as a label only, since the interaction is finished.*

Figure 5.12: *The sequence of updating a field of the Austrian prototype on the server through the client's user interface.*

hiding of interactive elements during times when they are not used, which is the case for the great majority of the time, avoids the disturbance of the controller's eyes with unnecessary GUI elements and on the other hand, since many of these fields are only changeable by specific users it helps them to figure out, which fields are editable by them and which are not. Figure 5.12 shows an example dialog if one of these interactive fields is edited.

Finally, one of the most complex dialogs we have implemented, while realizing these three prototypes, can be seen in Figure 5.14. The page, which is shown in Figure 5.11d, consists of a multitude of different tables to show radio frequencies for different airspace regions and flight levels. Different phases, i.e., sets of frequencies, can be created, edited, activated and deleted for each of these regions. The complexity of this task is not limited to the client part of the system, because, to make this complex behavior possible, many features and scripts had to be implemented on the server side as well. The interactivity level of this page and the user inputs that have to be stored in the server's database and distributed to other clients, exceed the other two prototypes by far. Nearly every action



(a) *Passive, with interactive elements hidden.*

(b) *Hovered and showing interactive elements.*

(c) *Passive, with interactive elements hidden.*

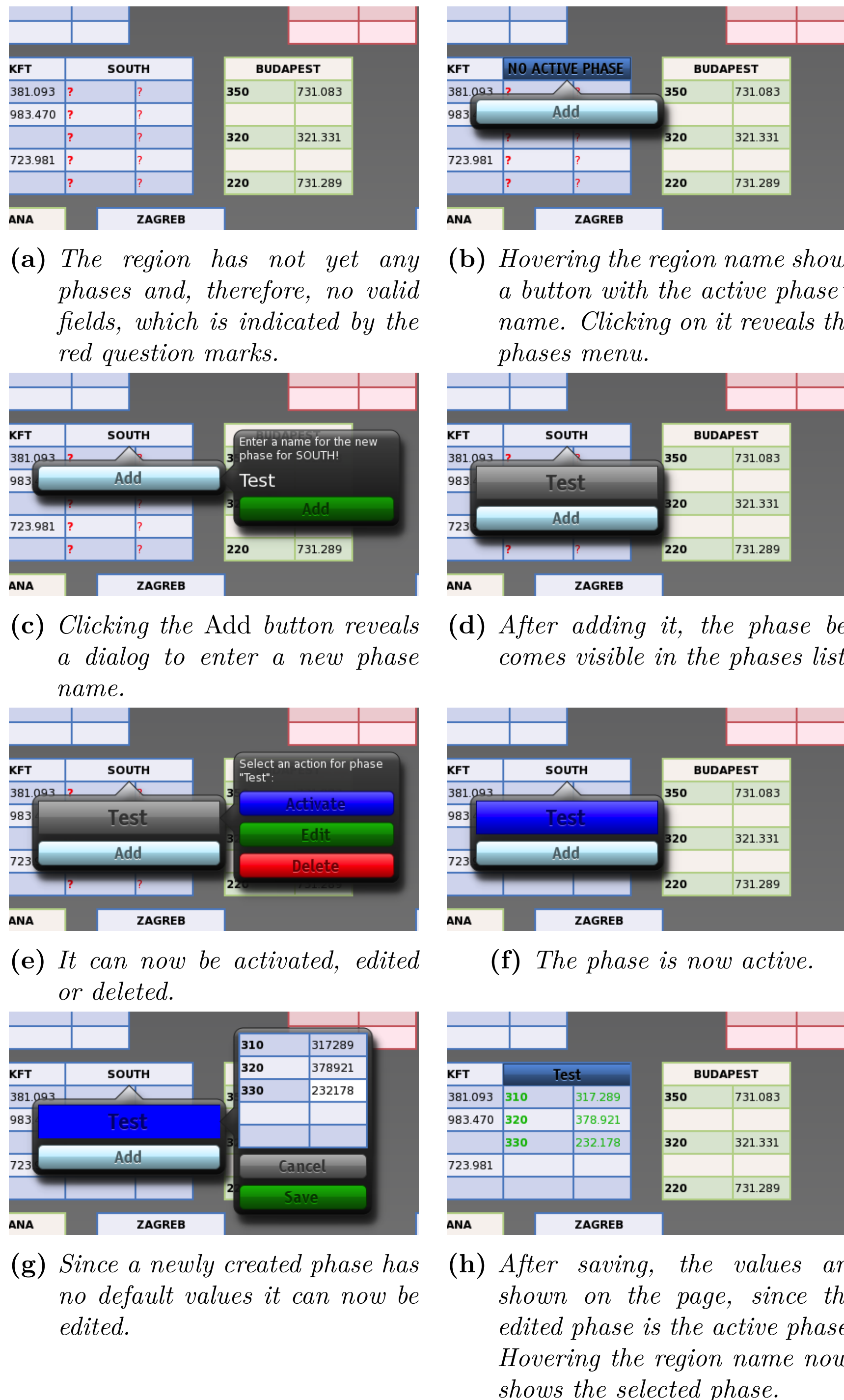
(d) *Hovered and showing interactive elements.*

Figure 5.13: *Interactive elements in these boxes are only visible if needed and do not disturb the user during normal operation. On the left side, the boxes are in their usual form, as informative displays. To the right, the boxes show their interactive elements if the user enters the box with the mouse.*

on the client that can be saved, is synchronized with the server and immediately available at every other client. This is not comparable to the typical systems presented before, where apart from airfield lighting controls, most of the application is centered around information consumption, with a very low level of possible inputs from the controller that have to be distributed.

Austrian Prototype: Implementation Efforts

When having a look on Table 5.3 it is interesting to see that even though this system differs fundamentally from the prototypes showcased before, the efforts are close to those spent for the second prototype. From a functionality point of view, this is a much more complex application, with behavior that is not related to the features of the aforementioned applications. Without knowing the numbers one would probably assume that the efforts are closer to prototype number one, since this was also the first application of its kind. We consider this fact a strong argument for the developed framework, since it is not only possible to develop standard applications, like the two prototypes before, but the system also allows us to create tailor made solutions for customers, without excessive efforts.



(a) *The region has not yet any phases and, therefore, no valid fields, which is indicated by the red question marks.*

(b) *Hovering the region name shows a button with the active phase's name. Clicking on it reveals the phases menu.*

(c) *Clicking the* Add *button reveals a dialog to enter a new phase name.*

(d) *After adding it, the phase becomes visible in the phases list.*

(e) *It can now be activated, edited or deleted.*

(f) *The phase is now active.*

(g) *Since a newly created phase has no default values it can now be edited.*

(h) *After saving, the values are shown on the page, since the edited phase is the active phase. Hovering the region name now, shows the selected phase.*

Figure 5.14: *One of the most complex menu workflows of the system. Everything happening in this sequence is updated in real time on the server, from the creation and activation to the editing of phases, all actions are distributed instantly to the other clients connected.*

Austrian Prototype: Controller's Comments

For this system we attended one workshop, where we presented the system in a very far developed state. There attended only controllers from Vienna and one military representative, no controllers from the LAUs have been available. In contrast to the Saudi Arabian workshops, we only showed a live demo of the system and discussed the inputs from the controllers together with them. In the following paragraphs, we present the results of these discussions.

We received very valuable input about usability issues, we have missed when developing the prototype. Interestingly, many of them are small issues that are easy to fix and they prove, how valuable it is, to show the system to the users in such workshops, especially with controllers involved that are already well educated in the use of software systems. One of the complaints, for example, is related to the frequency phases list shown in Figure 5.14. The menu hides the active values, which it must not do. Failed bulletin requests shall not result in a red striked out bulletin name but in a yellow one, a color-related topic similar to the one, that already came up for the Saudi Arabian windshear popup. The *Cancel* and *Save* buttons in the message boxes, like in Figure 5.10b, shall not hide input, since the controller wants to see the final layout of the text, before sending it. Also these message boxes shall use a monospace font to ease the layouting of these messages. The same font request applies for bulletins, since they are specially layouted as well.

Apart from these simple to fix issues there are as well more difficult ones that require more development efforts. One of these, is the request to forbid the input focus for all other elements, even in detached windows, if one message box already has the focus. Another one, is the request to change the layouts of the pages to avoid any empty space on them. If not the whole page is needed, the window shall shrink automatically to the size of the page's content. This requirement is a result of the target system where our system will be running, next to other systems from other vendors and the screen space is rather limited, so it shall not be wasted by empty background in windows.

5.5 Efforts Summary

As we see in Table 5.3 the numbers for the server and simulation adaptations show very similar efforts, independent of the scope of the project. What can be seen as well, is that the efforts clearly tend to be on the simulation side, with little adaptations needed for the server process itself. Regarding this outcome, the efforts needed on this side of the system will remain relatively stable for future projects and allow for easier cost calculations.

For the client side, we can see a tendency that the adaptation times are decreasing with each prototype, even if a custom system, like the third prototype, has to be implemented. These reductions are obviously related to a gain in expertise with the framework and the HMI technology in use. The efficiency may temporarily decrease, when people unused to the system get involved, when realizing new projects. A fact that has to be accounted in the calculation of related costs for these customers. In general, however, it can be

	hours	man-days
Client	213	28
Server	237	31
Not Assigned	268	35
Overall Efforts	**450**	**58**
Overall Efforts incl. N.A.	**718**	**93**

Table 5.4: *This table shows the efforts spent for a project, realized earlier and using a previously developed system, similar in purpose to the one proposed in this work but using older technology and a flawed, inflexible design. The efforts do not cover the implementation of the system itself, where unfortunately no details are available anymore. They only include the efforts needed for the adaptation of the system. The unassigned hours are not exactly accountable to either server, client or another task of the project. For this reason we include two overall values which results in a minimum to maximum range of possible overall efforts.*

safely assumed that the average adaptation times for the typical system will continue to further decrease for some time. This decrease will be directly related to the number of components that will be developed and made available for reuse.

Finally we also compare the adaptation efforts needed, using the framework created during this project, to the system AviBit has developed in the past. Since we do not have numbers on the original implementation efforts, we have to limit our analysis on these efforts. As can be seen in Table 5.4 there is an uncertainty in the numbers, since there are 35 man-days which are not directly accountable to neither client nor server. This means that even in the best case for the older framework, the efforts for the last two of the developed prototypes, are below the number of days needed to adapt the client to a new project. It is safe to assume that, as mentioned earlier, the first prototype needed more efforts due to the novelty of the framework and a lack of expertise with the technology used in the framework. If we consider that the former system only had limited, hardcoded simulation capabilities, which are not accounted in the numbers of Table 5.4, we can safely derive that the summed up efforts of at least around 60 days and at most around 90 days, for the old framework, are way beyond the efforts, needed for the new system. The latter consumes roughly around 25 to 30 days, to be adapted to a new project, simulation efforts excluded, as for the old framework. This means a minimum reduction of total efforts of around 50 percent, to adapt the new framework to a new project, compared to the former existing system. Time that can be used, to provide a fully simulated environment, that is not only useful during workshops but as well, during factory acceptance tests, where most of the interfaces are not available but the functional readiness of the system has to be tested, nevertheless.

5.6 Discussion

Considering the fact that the goal of this work, is a system that does not need a lot of code changes to be adapted to a new project, the numbers we have gathered show that we have accomplished this goal, since most of the efforts spent, can be accounted to

configuration tasks and the development of new HMI components. There will for sure be the need to add features to the framework itself at some point but from our experience during this project, this should be the exceptional case.

The prototypes with all their different features and the positive feedback from the controllers prove that the framework also has the necessary power, to create modern user interfaces and that it can be quickly adapted to user requirements. So far we have not come across any major technical limitations that would restrict our possibilities to realize user interface concepts and we have the chance, to extensively experiment with all user interface related topics, discussed in Chapter 2 (Related Work).

To be able to easily change HMI concepts, is also an invaluable advantage, when creating GUI designs for external interfaces, where the functionality it will deliver, is not yet fully known. The same applies for the powerful simulation framework, we have created. It lets us demonstrate and experiment with behavior, that cannot yet be tested with the final setup. The necessary interface integration, would otherwise be required, in early stages of a project, which is, most of the time, just not possible.

Despite the positive results, one of our system's limitations is that basically all graphical components, usually available in a GUI library, are not available in QtQuick. All of them have to be handmade, which on the one hand, solves the problem of a platform independent look and feel but on the other hand, increases initial efforts massively.

The framework's flexibility and scriptability, forms also one of its drawbacks. From our experience, it is sometimes very difficult to decide, whether to move functionality, from a project's specific configuration to the core framework, making it available for future projects as well. The way of handling this, that we propose, is to move such components to the core framework, only, if needed by another project as well. From our point of view, this is a reasonable trade-off, between maintenance efforts for modules and the risk to implement functionality twice.

One of the difficulties resulting from the use of QtQuick and a lot of scripting is related to automatic GUI testing, which is a rather complex topic on its own and even more so, if client server applications are in use. This is definitely something that has to be targeted in the future.

To summarize our results, we are very satisfied with the overall results of this work, since the framework proves to be the flexible system we intended to create. The framework also succeeds as a tool to communicate with controllers during workshops and the experience from this project proves, how important these workshops are, to finally ship a system that meets the controllers' expectations.

Chapter 6

Conclusion and Future Work

In this thesis we have presented a new approach of creating an innovative framework for ATC information systems. The main goal of the project, was the design, realization and evaluation of this system. In the following sections we summarize our results and discuss our plans for the future.

6.1 Conclusion

We have developed the core system and implemented three different prototypes with it. These prototypes cover three customers with 5 airports in Saudia Arabia, one Airport in Germany and basically all relevant airports in Austria, including area control centers and even military units. Each of the three systems integrates a different set of functionality. To be able to evaluate the project adaptation capabilities, we have tracked the efforts during the implementation period. During workshops we presented these prototypes to different controllers to receive their feedback. In the following paragraphs we summarize our findings on these activities.

- **Requirements Compliance**

  First of all, we summarize, to which level, the presented framework complies to the requirements that we have set up for it in Subsection 3.1.1 (Requirements and Goals):

  - **General Requirements**

    ✓ The design of the system, using a centralized server and individual interface processes, has proven to be functional and to easily integrate data from external systems.

    ✓ The client concept can visualize information in a wide variety of ways. New visualization methods can be easily integrated as individual components.

    ✓ Most of the system is adaptable through configuration and scripting, showing the value of the decision to use QtQuick as the base technology. This makes the framework ideal, to create prototypes for customers that can be improved iteratively, until system delivery.

  - ✓ The subscription based communication concept makes it possible, to easily make data available to interested systems. Access to the server is not restricted to the client system presented in this work; any application conforming to the protocol can receive data.

- ○ **Technical Requirements**
  - ✓ We have implemented the framework, following the rules we set, considering the API. The whole system is implemented solely based on the Qt framework, with the client's HMI built upon QtQuick, which together provide the required level of platform independence.
  - ✓ In Section 4.2 (System Design), we show the client- server-interface process architecture, we have designed for the system as required. Most of the data handling and processing is designed to be done on the server. We also comply to the thin client requirement, since the client mostly acts as a presentation entity, with controls and data modification entirely executed by the server or the relevant interface processes.
- ○ **User Interface Requirements**
  - ✓ We show in Chapter 5 (Results) that the framework allows for the creation of arbitrary custom graphical components that can then be integrated easily, by the declarative approach to create HMIs.
  - ✓ Since the framework facilitates full control over the used graphical elements, it allows for the implementation, of a platform independent look & feel.
  - ✓ The framework offers all of the different graphical possibilities to design user interfaces, discussed in Chapter 2 (Related Work). Colors and fonts can be used arbitrarily and animations and shapes can be created and integrated with ease.
  - ✓ The use of QtQuick enables the system, to be used on touch input based devices, since it offers relevant functionality, like finger gestures or kinetic scrolling, out of the box. GUI components can also be easily adapted, to better fit the needs of these devices by, e.g., increasing their size.

- **Adaptation**

  The following adaptation related results, have been found during the implementation of the prototypes:

  - ○ The client and server parts of the system can be adapted to a new project in about 25 to 45 days. The variance depends on the complexity of the customer's setup, including the share in HMI components that can be reused and the level of expertise of the developers, who are involved.
  - ○ To fully simulate the behavior of common external interfaces, another 20 days have to be included, in the efforts calculation. This estimation was true for all

three developed prototypes but will for sure vary if the number of interfaces differs significantly.

  - Prototype number three proves that the framework can be used to implement different variations of such information systems, because of the system's general purpose HMI capabilities.
  - Compared to an existing system, the new framework provides significant decreases in adaptation times.

- **User Reception**

  The system and prototypes themselves received generally positive reviews. However, there were also comments from the controllers to improve the system:

  - Controllers complained about usability issues. In the Austrian prototype, for example, some of the elements are not visible during interactions and covered by pop-up menus. This is unacceptable for them.
  - In two of the prototypes, the use of colors, especially of warning and alerting colors, is perceived differently by controllers with different backgrounds. In general, colors turn out to be the main, design related issue, for controllers. The use of animations, in contrast, has not been criticized in any way. This might be a result of having longer experience with colors, leading to faster responses and the relative newness of animations in such ATM systems.
  - For the third prototype, controllers requested user interface changes, related to the implementation of exclusive focus of input elements, to be sure that they to not forget, to send any relevant data.
  - Controllers in Saudi Arabia and Austria especially liked the idea to detach pages from the main window at will. They consider this, to improve their situational awareness through having more information available at the same time. Without this functionality, the page concept seems limiting to them.
  - We have also learned that screen space is often limited at the controller's working position. Therefore, it has not to be wasted by empty parts of an HMI. Instead, the window shall shrink to the necessary size, providing more place for other applications.
  - In general we have learned that the most critical point of controllers, is the fear of missing important information. This results in different wishes like the detachable windows, or the notifications through the highlighting of tabs, requested by the German controllers. In any case these fears have to be addressed.

Finally, the evaluation of the framework has proved that we have developed a valuable tool, to create information systems for the ATC business, with fast project adaptation times and a wide range of possible use cases.

6.2 Future Work

As the final part of this work, we point out future plans, we have for the framework, to improve it generally and to sort out the current issues, the controllers came up with:

- In general, we have to stabilize the use of colors, animations and GUI components throughout the framework, to achieve a better overall consistency in the user interface. In the same context, we will constantly extend a set of best practices, to avoid future user interface issues, as the ones we have discovered during this project.

- As one of the next steps, we plan to develop a general purpose notification mechanism for the client HMI, to manage the problem of controllers possibly missing information on a page that is not currently visible. Since, this is one of the biggest fears and situational awareness the most important goal, we seek to address this issue in an intelligent way.

- Another important and currently unsolved issue, is the integration of the system into an automated testing environment, which has to be one of the future investigations to prove and maintain the security of the system.

- Concerning the users, we plan to further improve the interaction between developers and controllers by stepping up our efforts, to involve the latter into the development process. From our point of view, a good solution might be, to let the controllers have access to the prototypes for a longer time than just the workshop periods. After these extended testing periods, controllers can then be interviewed personally or in written form, to evaluate their experiences and comments. Based on the results, the system could be better tailored to their needs.

- We also plan to further observe the development times needed, for future projects, since we assume that they will even further decrease, when the system evolves over time.

- In the future, with a potentially rising number of customers, there will be new developers working with the framework. On the one hand, next to providing an extensive documentation, we will evaluate the development of tools that further assist those developers in their configuration tasks. On the other hand, we will evaluate their performance, when executing these tasks, to see, where there is room for improvement in the framework itself. We also expect to create a graphical tool to debug the client's configuration, which at the moment has to be done through command line output.

- We plan to port the framework to Qt 5, once it gets stable enough, since it offers a further improved version of QtQuick, unleashing even more graphical capabilities. In this context, we plan to further experiment with the possibilities of user interface design principles.

- Currently all of AviBit's other systems have their own way of interacting with interface processes. As a long term goal, we plan to migrate all of these processes, to use the server that we have developed, as their centralized data source. To accomplish this, we have to provide the core part of the client as a library that can be integrated by these processes.

Acronyms

A-SMGCS Advanced Surface Movement Guidance and Control System. 10

ACC Area Control Center. 7

ACG AustroControl GmbH. 4

AFTN Aeronautical Fixed Telecommunication Network. 10

ANSP Air Navigation Service Provider. 1, 4

API Application Programming Interface. 31, 34, 37–39, 54, 86

ATC Air Traffic Control. 1–4, 8, 12–18, 20–22, 25–28, 34, 66, 75, 85, 87

ATCO Air Traffic Controller. 3, 10, 13, 14, 16, 21, 28

ATIS Automatic Terminal Information Service. 9, 22, 48, 66

ATM Air Traffic Management. 13, 14, 16, 17, 19, 21, 26, 87

COTS Commercial Off-The-Shelf. 16

DAQ Data Acquisition Process. 41, 42, 48, 49, 57–59

DFS Deutsche Flugsicherung GmbH. 4

FIDS Flight Information Display System. 71

FIR Flight Information Region. 7

GUI Graphical User Interface. 18, 20–22, 33–35, 37, 38, 41, 46, 51, 54, 55, 64, 79, 84, 86, 88

HMI Human-Machine Interface. 2, 3, 13, 15, 16, 18–20, 25, 26, 28–31, 33, 34, 37, 39, 41, 42, 44–47, 53, 55, 61–65, 68, 71, 72, 74, 75, 77, 82, 84, 86–88

ICAO International Civil Aviation Organization. 4, 43

IDE Integrated Development Environment. 38

IFR Instrument Flight Rules. 7

ILS Instrument Landing System. 7, 8

LAU Local Approach Unit. 75, 82

LLWAS Low Level Windshear Alert System. 10

M&C Monitoring and Control System. 39

METAR Aviation Routine Weather Report. 10

NDB Non Directional Beacon. 7, 11

PDF Portable Document Format. 64, 67

QML "Qt Modeling Language" or "Qt Meta Language" (both can be found). 34–38, 46, 47, 51–55, 74

SDK Software Development Kit. 34

SIGMET Significant Meteorological Information. 10

TAF Terminal Aerodrome Forecast. 10

TIS Tower Information System. 1, 26, 27, 35, 39

TMA Terminal Maneuvering Area (also Terminal (Control) Area). 6, 7

UI User Interface. 25

VFR Visual Flight Rules. 7

VOR VHF Omnidirectional Radio Range. 7, 11

WIMP "Windows, Icons, Menus, Pointer". 21

Bibliography

ACAMS AS [2013]. Product infos. Last Accessed: 17/01/2013.
URL: *http://www.acams.net/products.htm*

Athènes, S., Chatty, S. and Bustico, A. [2000]. Human factors in atc alarms and notifications design: an experimental evaluation, in *3rd USA/Europe Air Traffic Management R&D Seminar* [Fed 2000].

ATM [n.d.]. *USA/EUROPE Air Traffic Management R&D Seminars*, Federal Aviation Administration & EUROCONTROL. Last Accessed: 17/01/2013.
URL: *http://www.atmseminar.org/seminarPages.cfm*

Bertin, J. [1983]. *Semiology of Graphics*, University of Wisconsin Press, WI.

Cabrera, D., Ferguson, S. and Laing, G. [2006]. Considerations arising from the development of auditory alerts for air traffic control consoles, *12th International Conference on Auditory Display*, Vol. 12, International Community for Auditory Display, London, UK.

Cardosi, K. and Hannon, D. [1999]. Guidelines for the use of color in atc displays, *Technical report*, Volpe National Transportation Systems Center, Kendall Square Cambridge, MA 02142 USA.

CNN.com [2011]. Jumbo air france jet clips smaller plane at new york's jfk airport - cnn.com. Last Accessed: 17/01/2013.
URL: *http://edition.cnn.com/2011/US/04/11/new.york.plane.incident/*

Conversy, S., Gaspard-Boulinc, H., Chatty, S., Valès, S., Dupré, C. and Ollagnon, C. [2011]. Supporting air traffic control collaboration with a tabletop system, *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11, ACM, New York, NY, USA, pp. 425–434.

Digia [2013a]. Qt reference documentation: Signals & slots. Last accessed: 18/01/2013.
URL: *http://doc.qt.digia.com/qt/signalsandslots.html*

Digia [2013b]. Qt reference documentation: The arthur paint system. Last accessed: 13/01/2013.
URL: *http://doc.qt.digia.com/qt/qt4-arthur.html*

Digia [2013c]. Qt reference documentation: The meta-object system. Last accessed: 13/01/2013.
URL: *http://doc.qt.digia.com/qt/metaobjects.html*

Fed [2000]. *3rd USA/Europe Air Traffic Management R&D Seminar*, Vol. 3, Napoli, Italy.

Frequentis [2013]. Atm product infos. Last Accessed: 17/01/2013.
URL: *http://www.frequentis.com/de/at/solutions-portfolio/air-traffic-management/products-and-solutions/*

Graham, R. [1997]. Harmonisation of human machine interface the intuitive approach, *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, Vol. 2, pp. 6.3–10–6.3–17.

Hilburn, B. and Flynn, M. [2001]. Air traffic controller and management attitudes toward automation: An empirical investigation, *4th USA/EUROPE Air Traffic Management R&D Seminar*, Vol. 4, Federal Aviation Administration & EUROCONTROL, Santa Fe, NM, USA.

Jackson, A., Dorbes, A. and Pichancourt, I. [2000]. Striving for adequacy: The importance of rich hmi requirements, in *3rd USA/Europe Air Traffic Management R&D Seminar* [Fed 2000].

Kesseler, E. and Knapen, E. G. [2006]. Towards human-centred design: Two case studies, *Journal of Systems and Software* **79**(3): 301 – 313.

Leonidis, A., Antona, M. and Stephanidis, C. [2012]. Rapid prototyping of adaptable user interfaces, *International Journal of Human-Computer Interaction* **28**(4): 213–235.

MacKay, W. E. [1999]. Is paper safer? the role of paper flight strips in air traffic control, *ACM Trans. Comput.-Hum. Interact.* **6**: 311–340.

Mertz, C., Chatty, S. and Vinot, J.-L. [2000]. Pushing the limits of atc user interface design beyond s&m interaction: the digistrips experience, in *3rd USA/Europe Air Traffic Management R&D Seminar* [Fed 2000].

NAV CANADA [2013]. Navcansuite product infos. Last Accessed: 17/01/2013.
URL: *http://www.navcanatm.ca/en/navcansuite.aspx*

Ojanpää, H. and Näsänen, R. [2003]. Effects of luminance and colour contrast on the search of information on display devices, *Displays* **24**(4–5): 167–178.

Piazza, E. [2002]. Increasing airport efficiency: injecting new technology, *Intelligent Systems, IEEE* **17**(3): 10 – 13.

Rost, R. J., Licea-Kane, B., Ginsburg, D., Kessenich, J. M., Lichtenbelt, B., Malan, H. and Weiblen, M. [2009]. *OpenGL Shading Language*, 3rd edn, Addison-Wesley Professional.

Schlienger, C., Conversy, S., Chatty, S., Anquetil, M. and Mertz, C. [2007]. Improving users' comprehension of changes with animation and sound: An empirical assessment, *in* C. Baranauskas, P. Palanque, J. Abascal and S. Barbosa (eds), *Human-Computer Interaction – INTERACT 2007*, Vol. 4662 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 207–220.

Trim, R. [1990]. Mode s: an introduction and overview [secondary surveillance radar], *Electronics Communication Engineering Journal* **2**(2): 53 –59.

Tufte, E. R. [2001]. *The Visual Display of Quantitative Information*, 2 edn, Graphics Press, Cheshire, CT.

Ware, C. [2012]. *Information Visualization: Perception for Design*, Morgan Kaufmann Series in Interactive Technologies, San Francisco.

Weber, M. and Stone, M. [1994]. Low altitude wind shear detection using airport surveillance radars, *Radar Conference, 1994., Record of the 1994 IEEE National*, pp. 52 –57.