Marko Kivijärvi

CROSS-PLATFORM SOFTWARE COMPONENT DESIGN

Master’s Thesis
Degree Programme in Computer Science and Engineering
May 2013

ABSTRACT

The aim of this thesis is to analyze a project to design and implement a new FM Radio application for Symbian OS. The project process and relevant events are discussed when they have an impact on the design work. The goal of the project was to offer an improved user experience with features like favorite stations, song history, RT+ support, and automatic region selection. In order to complete the application within the project schedule, the existing radio modules were to be reused as much as possible. The application was required to be developed using the Qt application framework and to have no dependencies on the old UI framework from the Symbian OS. Platform-independence, testability, and simplicity were other key requirements for the new radio application. A final comprehensive goal was to stick to established design patterns and to follow the design principles and good practices defined earlier in the bachelor’s thesis by the same author. An added challenge was provided by the necessity to develop the application on top of a new UI framework that was still in development itself. Constant changes to the framework put a strain on the already tight project schedule. The discussion focuses on the design of the engine module that was required to house most of the application logic. The engine is disconnected from the Symbian OS by the use of a wrapper module, whose purpose is to hide the platform details. Due to its relevance to the engine, the design of the wrapper is discussed in detail. The application UI and the reused radio modules are discussed briefly and only to the extent that is relevant for the engine design.

The resulting design fulfills its requirements and the implemented application performs as designed. All the required features are supported, and the existing radio modules are reused. The lack of dependency on the old UI framework is witnessed by running the application in a system that does not contain the framework. The possibility to run the application on a Windows workstation also affirms that the platform-independence requirement was achieved. The design and implementation adhere to the principles outlined in the bachelor’s thesis and provide a practical use for them. The principles are found to be valid and important for the successful completion of a complex software project. It is discovered that the goal of designing a simple system from complicated beginnings is difficult to achieve and requires many iterations. Gradual refinements to the architecture and implementation are necessary to finally arrive at the optimal design. Constant refactoring is found to be a key element in the successful completion of a software project.

Key words: Qt, radio, engine, wrapper, Symbian


Avainsanat: Qt, radio, engine, wrapper, Symbian
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIIIVISTELMÄ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOREWORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFINITIONS AND ABBREVIATIONS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>1.1. Background and motivation</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>1.2. Roles and responsibilities</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>1.3. Objectives and scope</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1.4. Terminology</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>SOFTWARE DESIGN REQUIREMENTS</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>2.1. Use cases</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>2.2. Memory consumption</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>2.3. Performance</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>2.4. Testability</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>2.5. Simplicity and robustness</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>2.6. External dependencies</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>2.6.1. Platform dependency</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>2.6.2. UI dependency</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>HIGH-LEVEL DESIGN</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>3.1. Separation between UI engine and App engine</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.2. The UI engine</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.3. The wrapper</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>3.4. Layered architecture of the Symbian OS</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>THE UI ENGINE</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>4.1. RadioUiEngine</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>4.2. RadioStation</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>4.2.1. Implicit sharing</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>4.2.2. RadioStationPrivate</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>4.2.3. Shared null</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>4.3. RadioStationModel</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>4.3.1. RDS data reception</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>4.3.2. Special RadioStation instances</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>4.4. RadioScanner</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>4.5. RadioHistoryModel and RadioHistoryItem</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>4.6. RadioMonitorService and RadioControlService</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>4.7. Omitted architectural details</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>THE WRAPPER</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>5.1. Symbian implementation</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>5.2. Windows implementation</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>5.3. Logger</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>REUSE OF EXISTING RADIO COMPONENTS</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>6.1. Preset utility</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>6.2. Symbian-based app engine</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>NEW FEATURES</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>7.1. Automatic region selection</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>7.2. The frequency strip</td>
<td></td>
<td>39</td>
</tr>
</tbody>
</table>
7.3. Favorite radio stations .................................................. 41
7.4. RDS support ................................................................. 41
  7.4.1. AF field support ..................................................... 42
  7.4.2. PI field support ..................................................... 42
  7.4.3. PS field support ..................................................... 43
7.5. RT+ support ............................................................... 43
7.6. Song history .............................................................. 44
8. IMPLEMENTATION ........................................................... 45
  8.1. Discarded solution proposals ....................................... 45
    8.1.1. Threading .......................................................... 45
    8.1.2. DLL reduction ................................................... 46
    8.1.3. Parallel startup with radio server ......................... 46
  8.2. Temporary solutions .................................................. 47
    8.2.1. Radio settings ................................................... 48
    8.2.2. Media key handling ............................................ 48
  8.3. Memory management .................................................. 48
    8.3.1. Object lifetime management ................................ 49
    8.3.2. Smart pointers ................................................ 49
  8.4. Competitor comparison ............................................... 49
  8.5. Problems .............................................................. 50
    8.5.1. The problem with workarounds ............................. 50
    8.5.2. C++ templates with Qt ....................................... 51
    8.5.3. The constantly changing platform ....................... 51
    8.5.4. Time for refactoring ......................................... 52
  8.6. Testing .................................................................... 52
9. DISCUSSION .................................................................... 54
  9.1. Results ..................................................................... 54
  9.2. Reviews ................................................................... 55
  9.3. Windows version ....................................................... 55
  9.4. Difficulty .................................................................. 56
  9.5. The end of Qt radio and Symbian ............................... 56
  9.6. Future development ................................................... 57
10. CONCLUSIONS ............................................................ 59
11. REFERENCES ............................................................... 61
12. APPENDICES ............................................................... 63
FOREWORD

This master's thesis was written for the Department of Computer Science and Engineering in the University of Oulu under the supervision of Jukka Riekki. It covers the architectural design and implementation of a new FM Radio application for Symbian devices. The design and implementation was done in 2009 – 2010 while working as Package Owner of FM Radio for Nokia.

This thesis marks the end of a journey that took 15 years, 10 months, and 12 days to complete. Studying was put on hold for over 10 years while working as a software developer for various employers. The last 3 years was spent studying while working on a four-day workweek. Although delaying the studies for so long was not intentional, it was very enlightening and the work experience provided a good background for the remaining studies.

I would like to thank several people who helped me through the completion of this thesis. The first is my mother who gave me the idea for writing this thesis on radio design. I would also like to express my deepest gratitude to my supervisor Professor Jukka Riekki and second supervisor Professor Tapio Seppänen for their support and guidance. Thanks also belong to Heikki Pietilä from Nokia for reviewing and accepting this thesis. I was fortunate enough to work with Mikko Rahkila and Petteri Sipilä who created a pleasant working environment and later reminded me of project events that I had already forgotten. I am also extremely grateful to my good friend Veikko Tapaninen for his valuable review comments. Lastly I would like to thank my girlfriend Elina for putting up with me.

Oulu, May 2013

Marko Kivijärvi
## DEFINITIONS AND ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF</td>
<td>Alternative Frequencies</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>App</td>
<td>application</td>
</tr>
<tr>
<td>Avkon</td>
<td>UI library in the Symbian platform</td>
</tr>
<tr>
<td>Cell ID</td>
<td>base station identifier in the GSM network</td>
</tr>
<tr>
<td>DAB</td>
<td>Digital Audio Broadcasting</td>
</tr>
<tr>
<td>FM</td>
<td>Frequency Modulation</td>
</tr>
<tr>
<td>GSM</td>
<td>global system for mobile communications</td>
</tr>
<tr>
<td>HW</td>
<td>hardware</td>
</tr>
<tr>
<td>IAD</td>
<td>Independent Application Delivery</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunication Union</td>
</tr>
<tr>
<td>MOC</td>
<td>Meta-Object Compiler</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PI</td>
<td>Programme Identification (radio station identifier)</td>
</tr>
<tr>
<td>PS</td>
<td>Programme Service (radio station name)</td>
</tr>
<tr>
<td>Platform</td>
<td>operating system and its framework for running applications</td>
</tr>
<tr>
<td>PTY</td>
<td>Programme Type (radio station genre)</td>
</tr>
<tr>
<td>QML</td>
<td>Qt Meta-Object Language</td>
</tr>
<tr>
<td>QTestLib</td>
<td>unit test framework for Qt</td>
</tr>
<tr>
<td>Qt</td>
<td>application framework written in C++</td>
</tr>
<tr>
<td>RAII</td>
<td>Resource Acquisition Is Initialization</td>
</tr>
<tr>
<td>RDS</td>
<td>Radio Data System</td>
</tr>
<tr>
<td>RT</td>
<td>Radio Text (part of RDS to send text in 64 character blocks)</td>
</tr>
<tr>
<td>RT+</td>
<td>Radio Text Plus (standard to add identifying tags in the RT)</td>
</tr>
<tr>
<td>SIM</td>
<td>Subscriber Identity Module</td>
</tr>
<tr>
<td>SW</td>
<td>software</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>WSD</td>
<td>Writable Static Data</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Link Library</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

1.1. Background and motivation

In the fall of 2008, Nokia launched a large-scale project to create a new Symbian device family and to redesign all of the existing applications for it. A Norwegian software company called Trolltech ASA had already been acquired by Nokia on the 17\textsuperscript{th} of June 2008 [1]. Trolltech was acquired because it had developed an application framework called Qt, pronounced “cute” [2].

Qt is a cross-platform framework, and the applications using it can be compiled to any supported Operating System (OS) without changes to the code. The functionality of the applications is the same on all systems, and their appearance resembles the native applications in the system. The applications written with Qt will be platform-independent, meaning that they do not depend on any OS or OS-related framework. Qt hides the OS-specific details and provides a common Application Programming Interface (API) across all platforms. The Qt logo and slogan are shown in Figure 1.

![Qt logo and slogan](image)

Figure 1. Qt logo and slogan.

Nokia was planning to replace the old Symbian User Interface (UI) framework, called Avkon, with a completely new one by using Qt [3]. Avkon had won UI design awards in its prime, but had since fallen behind its competitors. The UI looked dated and was cumbersome to use. It was decided that Avkon with its heavy Symbian legacy was too complex to be brought up to par with the competition. The Avkon architecture was based on an ancient Psion architecture and extensive amounts of code was needed to create even the simplest of applications [4]. The Psion architecture had originally been designed for touch screen devices, but the touch screen support had been removed from Avkon at some point. Much later it was re-implemented, but the architecture had already evolved in a direction that could not naturally accommodate the touch screen.
A foundation had been formed for Symbian and its code was made public and open source [5]. Any third party developer was able to download the sources and to add features by contributing code to the foundation. However, many developers were shying away from the cumbersome UI framework and the peculiar look of the Symbian C++ code. Replacing Avkon with a modern, already established application framework would make adopting the Nokia application ecosystem much easier. Figure 2 shows the Symbian foundation logo.

![Symbian logo](image)

**Figure 2.** Symbian foundation logo.

A new UI framework, called Orbit, was being developed in Tampere using Qt. I was a member of a team that had just finished developing the Mobile TV application for Nokia N96 [6]. After some organizational shuffling, we became the new FM Radio team in the large application renewal project.

We were the new radio team that developed the new radio, but there was another radio team that was maintaining the old Avkon-based radio that would still be used in the older device families. Devices belonging to the older device families would still be made, and thus the old radio needed to be maintained and slightly upgraded. The old and new radio teams worked closely together, and the same Product Owner and Project Manager managed both teams. The old radio team consisted mostly of subcontractors, and they had plenty of experience in the radio area, whereas we had none. Coming from Mobile TV, the entire radio technology and its architecture in Symbian was completely foreign to us.

The project started in Tampere in a meeting where general guidelines and requirements for all applications were defined. The high-level UI renewal project was called "Timebox 10.1", indicating that the code-complete milestone was targeted for the first half of 2010. The commercial name for the new device family was to be Symbian^4, while the existing Avkon-based device family was rebranded as Symbian^3 [7].

### 1.2. Roles and responsibilities

Nokia was transforming its project teams from the traditional waterfall management model into the fashionable Scrum model, which is a form of agile project management style. Courses were taken, and books were read.
In the Scrum model, all team members have specific roles that specify their responsibilities in the team. The Product Owner prioritizes product features and essentially decides what should be done. The Scrum Master is a team leader, who ensures that the team can work as efficiently as possible and removes all obstacles that might slow them down. The team members do the actual designing, implementation, and testing.

Initially my role was a member of the development team, but eventually it was upgraded to Package Owner, which is Nokia’s addition to Scrum team roles. The Package Owner is the technical lead in the project and is responsible for the application architecture and quality. The term comes from the Symbian Foundation, where the software modules were called packages, and their owner was the one who knew most about the technical details and was responsible for the package functionality.

Since the beginning of the project, my responsibility was to create the architectural design of the application, and to be responsible for the code quality. The team also consisted of a Product Owner, Project Manager, Scrum Master, two developers, and a tester. The UI designer was not a member of the radio Scrum team but was still always available and present in daily meetings.

In addition to designing the architecture, the Package Owner was also a developer. After I had finished the architectural design, I started to implement it as a member of the development team. Due to unfortunate personnel resourcing issues in the team, I ended up doing roughly 80% of the implementation including everything mentioned in this thesis except for the test scripts. [8, 9, 10]

1.3. Objectives and scope

This master’s thesis aims to cover the design of the FM Radio application and the justifications for the decisions made in the design process. This thesis is a continuation to the bachelor’s thesis that was influenced by the same FM Radio project [11]. The bachelor’s thesis outlined various design principles and best practices for designing and implementing cross-platform software with Qt. The objective of this master’s thesis is to present the design and implementation decisions, and to analyze them specifically considering the principles outlined in the bachelor’s thesis. This thesis analyzes how well the principles were followed and how they benefited or hindered the design and implementation. Exceptional cases where the principles were not followed are also analyzed, and justifications for the decisions are offered. The analysis is performed after the implementation of the application has been completed and the results can be scrutinized.

The discussion begins in Chapter 2 with the requirements that were set for the application. The high-level application design is discussed in Chapter 3, and the engine design and implementation are discussed in Chapter 4. The wrapper plays a critical role in making the engine platform-independent, and therefore Chapter 5 has been dedicated to the wrapper. The reuse of the existing radio modules is briefly discussed in Chapter 6, and the new features that were added to the new radio are discussed in Chapter 7. Finally the implementation process and the issues that arose during it are discussed in Chapter 8.
The main focus is on the UI engine that is the most important component in the application. Other components are described only briefly when they have an impact on the engine design. Unit tests and automated tests are mentioned briefly but not described in any detail. The scope, as seen in Figure 3, covers the engine and wrapper and touches upon the UI and the reused Symbian components.

The thesis deals mainly with architectural decisions and the reasons behind them. Some implementation details of the components are also covered when they are relevant for the design. Similarly, some of the most important events in the project are also mentioned. The scope was chosen to highlight platform-dependency issues in the design.

![Figure 3. Scope of the thesis.](image)

### 1.4. Terminology

It is worthwhile to define a few terms that are used to describe various aspects of the FM Radio design. Early in the project, it was noticed that multiple names existed for a few key features and other radio related aspects. In order to reduce confusion, it was decided that the names should be made consistent and they should be documented.

The term “preset” refers to a radio station that is stored in the device. It is merely a collection of persistent data for a radio station, such as the name and frequency. The term “station” is used for a radio station in the device memory when the application is running. A station may be stored as a preset or it may simply exist in the memory for a short while. Presets may be pre-configured by the cellular network service provider, and when the radio is started for the first time, they are shown as
available stations. This thesis will deal with radio stations being manipulated in the device memory, and therefore the term “station” will mostly be used.

The term “frequency band” refers to the list of possible frequencies in a radio region. Different countries belong to different radio regions, and a radio application must only use the frequencies included in the frequency band. The frequency band spans from the minimum frequency to the maximum in frequency step increments specified by the radio region.

The term “tuning” refers to setting the receiver chip to a certain frequency. Although the receiver is not tuned as the old analog radios were, the term has still prevailed.

The term “seeking” refers to searching for available radio stations one at a time. To “seek up” or to “seek down” means searching up or down from the current frequency. The seek operation ends when the first audible station adjacent to the starting frequency is found.

The term “scanning” refers to seeking through the entire frequency band and storing all audible radio stations as presets. The scanning process begins from the minimum frequency in the frequency band and ends when the maximum frequency has been reached.

The term “application engine” or “app engine” refers to the reused Symbian-based engine module. In the old radio application, it had a similar purpose as the engine in the Qt-based application had, and due to the naming conflict, a new name had to be chosen. In hindsight, the separation between the engines is not stated clearly enough in their names. However, they were the names used in the project and they are therefore also used in this thesis.

The term “UI engine” refers to the new Qt-based engine in situations when the distinction between engines may be ambiguous. When no ambiguity exists, the Qt-based engine is also simply referred to as “the engine”.

The term “middleware” generally refers to any software component in the Symbian middleware layer, and in the context of this thesis, the term refers to the radio modules in the middleware layer.

The term “adaptation” refers to software modules that are responsible for adapting the device hardware chipset API for Symbian. The adaptation layer is responsible for abstracting away the details of the various hardware vendors.
2. SOFTWARE DESIGN REQUIREMENTS

The Product Owner and the architecture board in Tampere defined most of the design requirements. In addition, some personal requirements were set to ensure successful project completion and good code quality.

One of the key requirements set by the architecture board was to reuse existing Symbian-based modules as much as possible. Most of the applications that were being redesigned already contained a Symbian-based implementation and therefore had working modules that were to be reused if possible. For the radio application this meant that at least the radio components in the Symbian middleware layer were to be reused. The middleware modules were responsible for low level radio operations that manipulated the hardware through the adaptation layer. The reuse of the radio application engine in the application layer was also set as a goal. The reused radio components are described in Chapter 6.

The new radio application had to support all of the features supported by the old Symbian-based radio and much more. However, the first release of the application would limit the amount of additional features to ensure successful delivery. The remaining features would be added later as application updates. New features that did not exist in the old radio are described in Chapter 7.

2.1. Use cases

The engine use cases are illustrated in Figure 4. Light blue color is used to denote incoming notifications that must be handled by the engine, while the yellow color denotes activities that are offered by the engine.

Radio station management consists of the normal activities associated with radio stations. The user can add, edit, list, and remove them. The user can also tune to any saved station and skip forward or backward one by one. In addition, the user can make a phone call to the radio station and launch the web browser in order to view the station home page.

Audio control consists of the typical things that can be done to audio playback. The user can adjust the volume and mute or unmute it. Additionally the user can switch the audio routing from the headset to the loudspeaker and vice versa.

Radio tuner control consists of simple operations that are supported by almost all radio applications. The user can tune to any possible frequency, seek for stations one by one, or run a full scan of the entire frequency band.

Song history management consists of gathering the history, viewing and clearing it, and purchasing songs via the music store. Editing the history was not considered to be very useful, although the possibility of removing unwanted songs was considered.

The home screen widget support consists of sending status updates to the home screen widget and receiving commands from the home screen widget. The home screen widget is a small UI element that the user can attach to their device’s home screen. It can be used to monitor and control the FM radio playback. The widget
shows the station’s name, frequency, and Radio Text (RT). It also contains buttons to start and stop the radio playback and to skip between stations.

The engine can receive Radio Data System (RDS) data through the radio receiver chip. The data is received through the wrapper and it is shown to the user after preprocessing.

The engine can receive playback commands via the headset. In addition to the normal playback commands supported by the home screen widget, the headset can also control the volume level. The built-in volume buttons in the side of the device are handled as if the commands came from the headset.

The engine also receives device status updates through the wrapper. The signal strength notification could be used to intelligently detect abrupt disruptions such as going through a tunnel. Headset status change is used to route the audio to and from the headset. When the headset is connected, the audio is automatically routed to it. Finally the offline mode change is monitored because the radio may need to be closed when the offline mode is activated.

![Figure 4. Engine use cases.](image-url)
2.2. Memory consumption

One of the most critical requirements for any mobile application is to manage its memory consumption. Although the targeted device contained more memory than its predecessors, memory was still a sparse resource.

In practice, the memory management means that the application should use as little memory as possible, but more critically it means that no memory can be leaked. It was known in advance that a Qt-based application consumes more memory than a similar Symbian-based application because the Symbian framework is highly optimized for low memory devices. Proper memory handling was therefore a crucially important requirement for all Qt-based applications. [11 p. 26]

2.3. Performance

Optimizing the speed of the core use cases was an important requirement. It was considered crucial that the application started up quickly and that tuning to a station happened almost instantaneously.

In addition to the actual performance of the application, it was also considered important to optimize the perceived performance. This may mean drawing something quickly on the screen at the beginning of the startup sequence to indicate that the application responded to the launch command. This is especially critical for the radio application, since it has a slow hardware initialization process that needs to take place before the audio can be heard.

A special high-speed camera was purchased to allow observing the startup sequence very accurately. The expensive camera helps to remove flickering caused by unnecessary drawing that is barely noticeable with the naked eye. A very difficult target of 900 milliseconds was set for the perceived startup time. It was known that it would not be possible to get the hardware initialization done in that time, but the goal was to get the UI constructed and visible.

2.4. Testability

Testability was an important requirement set by the development team. Assuring good quality is impossible without thorough testing. The intent was to automate the unit tests as far as possible. The distinction between unit tests and module tests can be vague, and therefore the term “module test” was never used in the project and will not be used in this thesis. In addition to the unit tests, a test framework that was called “Matti” and later renamed “TestDriver” was to be used to implement the automated tests. Test-driven development (TDD) was discussed as an idealistic goal but was not officially required [11 p. 20].
The existing Symbian-based radio modules had no unit tests. The tests were rumored to have existed in the past, but for some reason they had been lost at some point. This meant that, although the existing components were said to be “tried and tested”, there was no way to verify that. The components obviously worked at least as far as the most common use cases were concerned, as could be seen from the millions of devices running them. The lack of unit tests was worrying, though, and it was decided that the tests needed to be implemented. The maintenance responsibility of the radio middleware modules was transferred from a team in Dallas to us, which also increased the importance of implementing proper tests.

To help the unit testing, the engine had to be compiled as a Dynamic Link Library (DLL), which could be easily loaded into the test module. The engine also needed to help the test framework by notifying it at certain checkpoints such as completed startup, for example.

2.5. Simplicity and robustness

A very important personal requirement was to keep the design and implementation as simple and robust as possible [11 p. 19]. This meant avoiding overly clever solutions. The design had to stick to established and well-known design patterns, and everything had to be easily understood by any reasonably knowledgeable developer.

The requirement to be easily maintainable is an automatic one that is typically set for all projects. However, as easy it is to write down in the list of requirements, it is surprisingly difficult to define. It was thought that simplicity is a prerequisite for maintainability, as well as being much easier to define and to communicate to others. The requirement list therefore mentions simplicity instead of maintainability, although in practice they both work towards the same goal.

In addition to maintainability, the simplicity requirement can also be seen as a prerequisite for extensibility. The ability to be easily upgraded and extended is also an automatic requirement that is much easier to understand in terms of simplicity. It was clear from the beginning that the first release of the application would not support all features that had been requested by the Product Owner. A release plan had been formulated to specify the order in which the new features would be introduced. In addition to the planned extensions, all successful designs must be prepared to accept new, unexpected requirements. Projects typically take years, and it is impossible to anticipate the shifts in focus and priorities in advance. The design must be able to easily adapt to changing situations instead of steadfastly maintaining the course that was set in the beginning. It was therefore crucial for the design to be naturally extensible, but again, the requirement for simplicity crystallizes it to everyone involved.

For the implementation, the simplicity meant that some C++ features were not to be used. In addition to the obvious things such as the “goto” statement, the switch-case statement was also not to be used unless necessary. Forever-loops and C-style casts were banned altogether. These implementation restrictions were made in an effort to anticipate and prevent future errors caused by careless mistakes. By anticipating the ways in which another developer may misunderstand the code and
accidentally break it, it is possible to lessen the likelihood of running into problems later. [11 p. 23]

2.6. External dependencies

One way of achieving simplicity and maintainability is to minimize dependencies on external components. Many dependencies on other components are vital, but unnecessary dependencies are merely a source of problems. The fewer external dependencies a component has, the less there are things that can go wrong in surprising ways.

One requirement, not set by the Product Owner or the architecture board, was to implement the application using a certain multimedia application framework. The upper management made this request for consistency purposes. It was considered beneficial if all media applications were implemented using the same framework and would therefore be able to share components with each other. However, no practical use case existed for such a situation that involved the radio application, so the requirement was rejected. The usage of the framework would have introduced a crippling dependency on it, as the framework dictated the designs for the views, engine, and application. By not using the multimedia application framework the radio was able to become much simpler and more independent. The upper management expressed their dissatisfaction, but the decision was not reversed.

2.6.1. Platform dependency

The application was required to be implemented with the Qt framework that Nokia had chosen as the application framework for mobile devices. The design had to hide the underlying Symbian OS in an effort to be as platform-independent as possible. The intent was not to discard Symbian, but simply to utilize the possibilities offered by the Qt framework as well as possible. The goal was to eventually shift from Symbian C++ to the more standard-like Qt C++ in the application layer.

To ensure that the goal of platform-independence would be met, a personal goal of supporting two platforms from a single code branch was set [11 p. 22]. The intent was to develop the application in a Windows workstation without running it in the Symbian emulator, which was too slow and cumbersome at the time. It had to be possible to compile the application for Windows and to run it natively. This required a new set of wrappers to be implemented for Windows. The Windows wrapper had to have an identical API to its Symbian counterpart and consist of dummy implementations for all operations.
2.6.2. **UI dependency**

Another personal design requirement was not to create a UI dependency in the engine. The engine would offer an intuitive interface to the UI layer, but it would not link against any UI library. Officially this meant that the entire application could not have a dependency on the Avkon UI framework, but a personal requirement of not having an Orbit dependency in the engine was also set.

As the radio application was being developed for Symbian, there was a separate team developing a radio application for Meego devices. Implementing two new separate radios seemed like a waste of resources, and made sense to attempt to eventually combine them. Instead of the Orbit UI framework that was being used in Symbian, Meego had its own UI framework called Direct UI. It was also implemented with Qt, which meant that as long as the engine did not have any UI dependencies, it would be possible to run the engine also on Meego.
3. HIGH-LEVEL DESIGN

On the high level, the application has been structured into layers that sit on top of each other. Only top-down dependencies are allowed between layers, meaning that a layer cannot make any assumptions of the layer above it. Each component has one simple responsibility as stated by the single responsibility principle (SRP) [11 p. 19]. The application is only responsible for starting the process and creating the engine. The UI module is only responsible for user interaction, the wrappers only hide the underlying layers, and the engine houses the application logic.

The observer pattern is used when messages need to be passed upwards from the lower layers. Signals and slots could have been used between the wrapper and the engine, but the traditional observer pattern was used instead. Using signals and slots incurs a minor performance penalty, and the typical problems of the observer pattern are not a concern in this case. The “observer” instance is set during application startup and never removed, which means that calling a function from a destroyed object cannot happen. [11 p. 16]

The classes and functions are named as logically as possible without using abbreviations. No attempt was made to be specifically terse, but instead, the names are as verbose as they need to be in order to avoid confusion and to be easily understood by all. [11 p. 21]

Figure 5, taken from the project wiki page, illustrates the high-level design of the radio application. The diagram is highly simplified, containing only the most relevant classes and dependencies. The diagram gives an overall view of the application. The details of the UI are not in the scope of this thesis.

A specific color-coding is used in the Unified Modeling Language (UML) diagrams in an attempt to make them as clear and easy to understand as possible. The actual colors were chosen more or less based on a personal feeling about the modules during the project work. Red color represents Symbian code that should be kept to a minimum, and green code represents pure Qt code. Orange color is used to represent mixed classes that contain both Qt and Symbian code. Finally, light blue color represents a dependency on the Orbit UI framework.

The color-coding was used in the design presentation made for the architecture review board. It was also used in A3-sized posters hung on the cubicle wall and in the offices of Product Owner and Scrum Master.
Figure 5. High-level design.
3.1. Separation between UI engine and App engine

The reuse of the engine module from the old Symbian-based radio caused a naming conflict between the two engines that needed to be resolved. The old and new radio applications were both designed using the Model-View-Controller design pattern, and both had a controller called “the engine” [11 p. 21]. Both engines had the same purpose in their respective applications, but the new Qt-based engine was meant to use the services offered by the old Symbian-based engine in the new application.

The naming conflict was resolved by renaming the Symbian-based engine the “app engine” (Radio Engine in Figure 5) because it was the application engine in the Symbian-based application. The Qt-based engine was named the “UI engine” (Radio UI Engine in Figure 5) because it is the engine that serves the UI layer and adds new features on top of the app engine. In hindsight, however, the distinction between the names is not clear enough, but they were used during the project and therefore, they are also used in this thesis. The UI engine is sometimes simply referred to as “the engine” in this thesis. The reused app engine is discussed only briefly in Chapter 6.2.

3.2. The UI engine

The UI engine is the most independent module of the application, and it is the intended location for the application’s “business logic”. This means that the main decisions involved in various features are made in the UI engine. By simply counting the lines of code, the biggest bulk of a given feature often resides in the Symbian-based components at the lower levels. However, the feature is offered as a collection of separate services through the component API. The UI engine is the place where these enabling services are tied together to form the complete feature that is offered to the UI layer.

The engine is loaded by the application during startup, and it assumes responsibility of all radio operations. It depends only on the Qt framework and the wrapper. The dependency on the Qt Highway framework for communicating with the home screen widget was intended to be temporary and planned to be replaced by the new Qt service framework once it was available.

3.3. The wrapper

The purpose of the wrapper is to hide the platform-specific implementation, and to expose a consistent API to the Qt-based engine. The wrapper actually consists of two modules. The “Engine wrapper” hides the app engine, while the “Radio Preset Storage” hides the preset utility module and follows the same design principles as the engine wrapper. The preset storage wrapper is also extremely simple, and therefore it is not discussed separately. The detailed discussion in Chapter 5 will handle the
engine wrapper and the preset storage as one unit and simply refer to them as “the wrapper”.

In order to ensure proper platform-independence and ease the development, the wrappers were developed for two operating systems: Symbian and Windows [11 p. 22]. The Symbian implementation passes data and commands between the Qt-based engine and the Symbian-based modules and does all the necessary type conversions. The Windows implementation is only meant to simplify implementation and testing, and it acts as a stubbed dummy engine.

3.4. Layered architecture of the Symbian OS

The architecture of the Symbian OS is separated into specific layers, which have strict dependency rules. The dependencies can only go top-down, meaning that the lower levels cannot depend on the upper ones. The majority of the radio application modules reside in the application layer.

Figure 6 illustrates the layering in the Symbian OS and the placement of different modules of the radio application. The middleware layer is not covered in this thesis except for the refactoring of the preset utility module discussed in Chapter 6.1. The figure uses the same color-coding that was used in Figure 5.
4. THE UI ENGINE

The first notable thing about the engine architecture is that all API classes utilize the pImpl design pattern to hide the implementation details [11 p. 15]. Only the API is exposed to the users of the engine, and everything else is hidden. This decouples the UI from the engine. The engine is built as a separate DLL to help unit testing and to clarify the separation from platform-dependent modules [11 p. 19].

The engine offers the supported features via its API functions and properties. All API classes have a private counterpart behind the pImpl pointer, called the d-pointer in Qt, to hide the internal implementation. The pair consisting of public and private classes forms a single unit of responsibility, and therefore the following discussion will handle them together. RadioStationPrivate is an exception that deserves a chapter of its own, but the other private classes will not be singled out.

Figure 7 shows a simplified view of the internal design of the engine module. Color-coding has been used to highlight various classes in the design.

![Radio engine design](image-url)
4.1. RadioUiEngine

RadioUiEngine is the primary API class of the engine. During startup, the radio application executable creates an instance of RadioUiEngine, and then calls a special `init()` function which performs the rest of the startup procedure. Instead of performing the startup procedure in the RadioUiEngine constructor, the two-phased construction with the `init()` function is used to make error reporting easier. If the startup fails for some reason, the application can ask the RadioUiEngine instance for contextual information. The other alternative for startup error handling would have been using exceptions, but for the sake of simplicity it was never seriously considered.

Although the use of singletons is generally not recommended, the RadioUiEngine is a singleton [11 p. 21]. Almost all classes in the engine need a pointer to RadioUiEngine, and having the engine available as a singleton simplifies their construction. As a rule, the classes can only get the singleton instance during their construction, although the rule cannot be easily enforced in code. They save it as a member variable that they can use later. No class is allowed to get the singleton instance during normal operation.

4.2. RadioStation

RadioStation is a central class in the radio engine design. It is a simple data holder class, whose sole purpose is to store radio station data in memory. Almost all classes in the engine have a dependency on RadioStation, whereas RadioStation does not depend on any engine class. These dependencies have been omitted from Figure 7 to keep it from becoming too cluttered.

The data is grouped into two categories. The first category contains persistent data that must be saved to disk in order for the data to be available after restarting the application. Persistent data contains information such as station frequency, name, home page, PI code, and genre. The second category contains non-persistent data that is valid for a short period of time and should not therefore be saved. Received RDS data and various computed values are examples of non-persistent data.

When a radio station sends RDS data, the data is stored in a RadioStation instance that corresponds to the currently playing station, and the instance is sent as a parameter in a signal notifying about the received data. All data signals have a RadioStation instance as their single parameter, which makes handling incoming data easy. The signal can be connected to any number of slots depending on the use case, and each slot can get the relevant information from the RadioStation instance that is passed by value [11 p. 28].

The RadioStation was declared as a Qt meta-type by using the special `Q_DECLARE_METATYPE` macro to allow it to be put inside a QVariant, which is a container for generic data. This enables the RadioStation to be passed through the QAbstractItemModel API functions that use QVariant to handle data. The RadioStation header file is presented in Appendix 2. It shows the declarations of the topics mentioned in this chapter.
4.2.1. Implicit sharing

RadioStation uses implicit sharing to make it cheap to copy and use as a signal parameter [11 p. 28]. Implicit sharing allows it to be passed around by value without having to worry about performance issues or memory leaks. Functions that search for radio stations from the model can safely return the found instance by value. The returned value will not point to a stack-based variable that will go out of scope, and even ignoring the returned value will not result in a memory leak. The compiler will take care of freeing the returned instance if it is not used.

Implicit sharing also minimizes the overall memory usage for radio station data. The radio application supports a maximum of 100 saved radio stations, and at any given time, there may be multiple copies of some of them in memory. Although the data duplication with the station instances would not amount to a very significant memory waste, it is still a waste, and implicit sharing minimizes it.

4.2.2. RadioStationPrivate

The RadioStation class defines the publicly visible part of implicit sharing, and RadioStationPrivate is responsible, as the name suggests, for the hidden private part. It contains member variables for all of the radio station data elements, and it also contains a reference counting mechanism. Instances of these private data holders are shared between RadioStation instances. When a RadioStation instance attaches to the private class, the reference count is incremented, and similarly the reference count is decremented when the public instance detaches from it. If the reference count reaches zero, the data instance will be destroyed.

Attempting to modify the data, for example calling a function to set the station name, will cause a special operation to take place before the modification can be done. If there is only one reference to the data, the public class has sole ownership of it, and the modification can be done without further actions. If, on the other hand, the reference count is higher than 1, there is another RadioStation instance pointing to the same data, and modifying it directly is not allowed. The public class will first detach from the data, make its own copy of it, and then modify the new copy leaving the original data untouched.

RadioStationPrivate also hides an implementation detail not related to implicit sharing. As will be described in Chapter 6.1, the only way to offer persistence to radio stations was to save them via the preset utility. The preset utility is a normal Symbian module and therefore cannot directly handle RadioStation instances that contain Qt code. Furthermore, the RadioStation cannot have a dependency on Symbian because the engine needs to be kept platform-independent. In order to achieve layer separation the RadioStationPrivate derives from an interface called RadioStationIf that is defined in the wrapper. RadioStationIf contains declarations of pure virtual functions to get and set radio station data, and RadioStationPrivate provides an implementation for them. A special convenience function exists in RadioStation to cast the RadioStationPrivate to RadioStationIf, which is used by the RadioStationModel.
4.2.3. Shared null

Shared null is an idiom that increases application performance by making the default initialization of complicated objects very cheap. Shared null is a special instance of implicitly shared data that is defined to exist statically in memory. Creating new instances of classes that utilize the shared null idiom requires no data variables to be initialized, because the public class can simply reference the shared null instance. When the created instance attempts to modify the data, it is detached from the shared null instance, as is usual with implicit sharing.

Radio uses shared null by specifying a static instance of RadioStationPrivate that is referenced by all new RadioStation by default. As the amount of data stored for radio stations, as well as their complexity, is very low, the performance benefit is negligible. It is used merely as a curiosity, as it does not complicate the design or cause any other harm.

4.3. RadioStationModel

RadioStationModel manages the list of radio stations saved in the application. It derives from QAbstractListModel, which allows it to be used as a data source in a list control in the UI.

RadioStationModel maintains a list of RadioStation instances in memory and organizes their storage through the wrapper layer. Implicit sharing requires that users of the implicitly shared class should not have direct access to the data, because the actual data class instance may change when the public class detaches from it and creates a copy. RadioStationModel breaks this rule when it reads or writes radio stations through the wrapper by accessing the RadioStationPrivate instance directly. This was considered to be acceptable because there can be no other references to the data when stations are being read from the disk, and other references are not affected when they are being written.

4.3.1. RDS data reception

RadioStationModel is responsible for receiving RDS data, saving it to the appropriate RadioStation instance, and reporting to the UI about it. The private class derives from RadioStationHandlerIf defined in the wrapper to receive the RDS data. Successful RDS data reception sometimes requires special logic because of limitations in the RDS implementation in the radio middleware, as discussed in Chapter 7.4.

Figure 8 illustrates the sequence of receiving RDS data. The sequence diagram has been simplified by omitting components such as the radio server client and RadioStationModelPrivate. The sequence shows how the data is saved in the
RadioStation instance, which is then sent as the signal parameter to the UI. The sequence diagram uses the same color-coding that was used in Figure 5.

Figure 8. RDS reception.

4.3.2. Special RadioStation instances

RadioStationModel also maintains special RadioStation instances called the “current station” and the “manual station”. Internally they do not differ from other RadioStation instances, but they have a special purpose in the radio engine.

The “current station”, as its name suggests, is actually only a pointer to the RadioStation instance corresponding to the currently tuned frequency. The pointer is updated whenever the engine tunes to a new frequency. The pointer is dereferenced when received RDS data is saved, and also when any component asks for information about the current station.

However, the user is able to tune to any possible frequency in the frequency band, even to ones that are not saved in RadioStationModel, and therefore have no corresponding RadioStation instance. A typical use-case is a user travelling to another city, and instead of running a full scan, simply manually tunes to a frequency. Information about such a manually tuned station is saved in the special
“manual station” instance. When the user is listening to a manually tuned station, the “current station” pointer points to the “manual station” instance. This arrangement ensures that no data is lost if the user decides to save the station after listening to it for a long time. All received data is kept in the manual station instance, and can be saved into the model. When the user tunes to another manual station, all received data is removed.

4.4. RadioScanner

RadioScanner is responsible for handling the process of scanning for available radio stations. Scanning is a relatively complex process that requires extensive handling in all layers. The majority of the implementation resides in the radio middleware and adaptation layers, which actually do not have any concept of scanning the full frequency band. The middleware contains API functions for tuning to a desired frequency and for seeking up or down until an adjacent station is found. The RadioScanner uses these two functions to perform the full frequency band scan. First it calls the tune function to tune to the minimum frequency in the frequency band, and then it proceeds to repeatedly call the “seek up” function until the entire frequency band has been covered.

Due to unfortunate design limitations in the radio middleware, the situation when no audible stations are found needs special handling. In this situation, the seek operation begins from the minimum frequency, finds nothing from the entire frequency band, and loops round back to the beginning, where it stops. The middleware has no way to report that this has happened. It simply reports the minimum frequency, and the RadioScanner must be able to tell that the seek operation has looped and the reported frequency should not be saved.

All found radio stations are saved via RadioStationModel. The scanner reports the found radio stations with a signal that provides the newly created RadioStation instance as a parameter. The UI uses this signal to update the scanning progress indicator and to draw the appearing stations into the UI.

When the scanning operation begins, the scanner tells RadioStationModel to prepare for scanning. RadioStationModel moves all radio stations into a temporary memory array for the duration of the scanning. When a station is found and the reception is good, the PI code can be received before the scanner proceeds seeking for the next station. If the PI code matches a station in the temporary array, it is reused. This is done to ensure that no data is lost for stations that were already saved in the model. The temporary array is cleared after scanning, which means that inaudible stations are removed.
4.5. RadioHistoryModel and RadioHistoryItem

RadioHistoryModel and RadioHistoryItem are responsible for storing information about played songs in a database. The user is able to scroll through the song history and to purchase songs through the music store. The first release of the application was intended to contain only a minimal support for song history. The later versions were planned to contain more features, but it was considered useful to at least have the model collecting data in the first release. This was intended to make the feature adoption more pleasant, when the model would already contain song data.

Architecturally the RadioHistoryModel and RadioHistoryItem do not bring up any new design considerations, as their structure is very similar to RadioStationModel and RadioStation. Detailed architectural discussion would be repetitive, and it is therefore omitted.

4.6. RadioMonitorService and RadioControlService

RadioMonitorService and RadioControlService offer an API for a home screen widget to monitor and control the radio application. The home screen widget is a simple remote control UI for the radio application that can be placed in the home screen. The widget UI consists of simple text elements that show the station name and RT as well as buttons to control the radio playback.

The widget runs in the home screen process while the radio application runs in its own process, which means that an Inter-Process Communication (IPC) mechanism is required to pass messages between the processes. Qt service framework was not available at the time of implementation, so a temporary component called Qt Highway was used to pass the messages. The technical details of the IPC mechanisms and therefore of RadioMonitorService and RadioControlService are not in the scope of this thesis.
4.7. Omitted architectural details

The design description of the engine has been simplified by omitting some features that do not bring up any new design considerations. The following engine details are therefore ignored completely and not discussed in any detail.

*RadioFrequencyModel* is a dedicated model to help UI controls show valid frequencies in the frequency band. It contains all possible frequencies in the frequency band, whereas *RadioStationModel* contains only the saved radio stations. It is an important component for the UI, but as far as the engine architecture is concerned, it is trivial and uninteresting.

*RadioTimerPool* is responsible for managing timers needed by various radio components. A component can request a timer to fire after a given delay and provide a slot to be called. The purpose of the pool is to simplify timer handling. Classes that use timers rarely do not need to have a member variable for a timer, when they can simply ask for a timer instance from the pool. The design and implementation of the timer pool is trivial, and it is therefore omitted.

One of the requirements was to support a third party song recognition application that effectively “listens” to the radio broadcast, and attempts to recognize the song. Played songs were to be primarily recognized from the RDS data, as described in Chapter 7.5, but it was not supported by many radio stations at the time. It was decided that a third party application could be used to enhance the song history experience. The support for the third party song recognizer is an entire topic on its own, but it is not relevant for the engine design and is therefore omitted.
5. THE WRAPPER

The sole purpose of the wrapper module is to hide the underlying OS from the platform-independent engine. The wrapper is based on the wrapper design pattern, which is often referred to as the “adapter pattern” [12]. The name “wrapper” is used instead of “adapter” because the main purpose of the module is not to adapt anything, but to wrap and hide the engine.

Internally the wrapper uses the pImpl design pattern with public and private classes to ensure that none of the platform details leak through the API [11 p. 15]. As the wrapper needs to be implemented for every supported platform, it needs to be as simple as possible. The wrapper therefore contains no application logic and makes no decisions. It merely reformats data and passes it between Qt and the underlying OS.

The wrapper performs all necessary data conversions from Symbian descriptors to Qt strings and vice versa, and it also maps enumerated values to new enums defined in the wrapper. Redefining all enums in the wrapper ensures that the engine does not need to include platform-dependent header files.

The reused Symbian-based app engine, described in Chapter 6.2, consists of three DLLs, which are all accessed through the wrapper API. The wrapper is a Qt module that additionally contains Symbian code, which makes it an orange module in the color-coding convention of the high-level design. Figure 9 illustrates the internal design of the wrapper. The figure uses the same color-coding convention that was used in Figure 5 to show which classes have mixed Qt and Symbian code.

![Figure 9. Radio engine wrapper design.](image-url)
5.1. Symbian implementation

RadioEngineWrapper is the main class of the wrapper, and it is instantiated by the engine during application construction. It has a private counterpart that houses most of the implementation of the class. The interface consists of functions to control the volume, frequency, and RDS reception. The wrapper is not allowed to have a dependency on RadioStation, which resides in the engine module above the wrapper. Therefore, the wrapper only supports tuning by frequency, not by station id.

RadioEngineWrapperObserver is an interface that follows the observer design pattern. It is used to notify the engine of radio events such as incoming RDS data and tuning notifications. The only observer, namely the engine, is added during construction and never removed. The observer pattern is used only as a way to decouple the engine from the wrapper.

RadioSettings stores settings such as the last tuned frequency in the Symbian Central Repository through the MRadioApplicationSettings interface and the CRadioSettings class in the reused settings module from the app engine.

RadioStationHandlerIf is an interface that the wrapper uses to notify RadioStationModel when RDS data arrives and scanned frequencies should be added as stations. It follows the observer pattern, and the private class of the model is its only observer.

RadioEngineHandler creates and owns the reused Symbian-based app engine. Notifications such as tuned frequencies and signal strength are reported through the MRadioEngineHandlerObserver interface to the private class of the wrapper where they are sent to the UI engine.

RadioRdsListener registers itself as an observer of incoming RDS data through the MRadioRdsDataObserver interface in the app engine. After converting the data from a Symbian descriptor to a QString the listener sends the data to the UI engine via the RadioStationHandlerIf interface.

RadioControlEventListener registers itself as an observer of control events through the MRadioControlEventObserver interface in the app engine. Control events are commands sent from the headset. The user can adjust the volume, control the playback, and skip to the next or previous station. This interface is used for legacy reasons described in Chapter 8.2.2 and it is only temporary.

5.2. Windows implementation

The purpose of the Windows implementation is to provide a working “dummy” that the engine can use as if it were connected to a real radio server and a receiver chip. It can be used during the development of the engine and UI.

The Windows implementation is very similar to the real Symbian implementation. They are compiled from the same project file, which chooses different private classes depending on the target platform. For example, the wrapper’s main class RadioEngineWrapper is shared between the Symbian and Windows implementations, but its private counterpart RadioEngineWrapperPrivate has a separate implementation for each platform. The project file chooses the file
“radioenginewrapper_p.cpp” when compiling for Symbian and “radioenginewrapper_win32_p.cpp” when compiling for Windows.

The Windows implementation mimics the behavior of the Symbian implementation as closely as possible. All asynchronous operations, such as tuning to a frequency, are also asynchronous in Windows, which is very important. A typical mistake when implementing a stubbed asynchronous request function is to be lazy and simply call the observer callback directly from the request function. This makes the operation synchronous, and it will behave differently from the real request.

There is no app engine in Windows, which means that the radio stations cannot be saved to Central Repository. They are saved into the Windows registry instead. To simulate various operations, such as frequency scanning and RDS reception, hard-coded values are reported from the wrapper.

The windows implementation declares a global pointer to the private implementation to offer an API for a test UI window. When the radio application is launched in Windows, a special test UI window is created, and the normal radio UI is created inside it. The test UI window has a test button console at the top. When a button is pressed, the test UI calls the appropriate function in the wrapper private class through the global pointer. Figure 10 shows the control flow when the “Add song” button is pressed in the test button console (see Appendix 3). The test UI accesses the wrapper through the global pointer and calls a test function. The function calls the RDS reception callback, and the engine processes it as if it were normal RDS data that arrived through the receiver. The UI and engine contain pure production code and are not in any way aware of the data being simulated. The dummy wrapper therefore provides a test bed for the application.

![Figure 10. Test UI control flow for testing received RDS.](image-url)
5.3. Logger

All classes in the wrapper are meant to be very simple, but the logger is an exception to the rule. As far as loggers go, it is very sophisticated.

Having detailed trace logs is essential for application development and debugging. The logs need to be easily produced, and they need to contain enough detailed information to be able to accurately report what the application has done. On the other hand, the logs should not get cluttered with too much information, and they should not contain anything unimportant. In addition, the possibility to change the logging level from basic to verbose is ideal when debugging a difficult problem.

A decent file logger was needed in order to see what the application was doing in the device when it was not connected to a computer where it could be monitored. A logger was needed for the Qt-based modules as well as the Symbian-based ones.

I had already developed a convenient file logger for the Mobile TV application that I had worked on earlier. The foreword of the article about the logger in Nokia Developer Wiki is shown in Appendix 1. The simplest use case for the logger is adding a macro called “LOG” into the beginning of a function, which causes the logger to trace the function entry, exit, and a thrown exception. The logger finds the function name and adds it to the log, and it also illustrates function call hierarchies by incrementing and decrementing the text indentation. A bit of assembly code is used to dig out the leave code when something has gone wrong in the function and a leave (exception) is thrown. The logger writes traces to a file in the device as well as to the standard error output. The file logger was taken into use in the Symbian-based app engine.

The complete logger mechanism is a two-part system. It was necessary to get the UI engine logs into the same file as the app engine logs. Otherwise it would have been very difficult to track which UI engine operation triggered which app engine operation. To get the logs in the same file, the Qt-based logger gets a pointer to the Symbian-based logger through the wrapper API, and sends all traces to it. The Qt-based logger supports almost all of the same features as the Symbian-based one, except that leave codes do not exist in Qt, and exception details cannot be retrieved. It therefore only reports that an exception has occurred.

The loggers are not normally included in release builds, but to maintain binary compatibility in the wrapper DLL, the function to get a pointer to the Symbian-based logger has to remain even in the release builds. When the logger is not included in the build, the function will return zero.
6. REUSE OF EXISTING RADIO COMPONENTS

It was recognized from the beginning of the project that there would not be enough time to implement everything "from scratch". It was decided that some components from the old Symbian-based radio application would be reused in the new one.

This was a natural time to do some refactoring, so the components were thoroughly reviewed. Some features were removed, some were added, and some were modified to better serve their purpose.

6.1. Preset utility

The preset utility is the module in the radio middleware that stores saved radio stations, i.e. presets. The presets are saved in a Central Repository file, and a maximum of 100 presets can be saved. This is less than the maximum amount of possible radio stations in a frequency band, but it was considered to be enough.

The Central Repository is the standard component for saving application settings in Symbian. A Central Repository server handles compressed files that reside in the application’s private directory, and offers a client API that handles simple key-value pairs. The basic data types are supported, and the preset utility uses the string data type to save radio presets by serializing and de-serializing them to and from the string.

When the project was started the preset utility was reviewed and found to be in dire need of refactoring. New data elements had to be added for each preset to support the new features. Also, the original design of the preset utility did not allow for future expansion without breaking binary compatibility. This was seen as a critical problem, as the likelihood of having to add new data elements to radio stations at a later time was estimated to be high.

Since the preset utility is a simple component, it was decided that a complete redesign was the easiest and fastest solution. As far as the new Qt-based radio was concerned, it would have been best to implement the new preset utility as a Qt module, but that was not possible.

The radio maintenance team was doing a facelift for the old Symbian-based radio, whose presets also required new data elements. This meant that the new preset utility had to work with both radio applications, and therefore it had to be implemented as a Symbian-based module. Another factor was the fact that porting Qt to Symbian was not yet complete and Central Repository was not yet exposed to Qt.

The new preset utility was designed to be easily extensible without breaking binary compatibility. The preset data is stored in a base class that can be extended with new data elements.

One additional thing that needed to be considered with the preset utility was the possibility of supporting an Independent Application Delivery (IAD) update with the Symbian-based radio. In an IAD update, the user is delivered an installer package that updates the old application to a newer one, while the rest of the system remains unchanged. It is important that the user does not lose any data in an update, which posed a problem since the old and new preset utilities were not compatible. When the
user starts the radio for the first time after the update, the previously saved radio stations should still exist. The problem is that the new version of the preset utility is unable to read the Central Repository file written by the old version, as the data structure has changed.

It would have been possible to add a data migration capability to the new preset utility, but the simpler and safer solution was to include the new version of preset utility in the N8 device that had not shipped yet. The N8 was the first device in the Symbian^3 device family, and if all the devices contained the new version of the preset utility, there would be no issues with losing data in an update. In order for the Symbian-based radio application in N8 to use the new version of the preset utility, it would have needed changes to its code, but such a big modification near the shipping date was rejected. Instead, a small adapter DLL was implemented to adapt the new preset utility API to the old one [12]. The name of the adapter DLL is identical to the old preset utility, as is its API, which meant that no changes needed to be made to the radio application code. After the IAD update, the adapter would simply be left unused, as the updated radio would be modified to use the new API directly.

6.2. Symbian-based app engine

The app engine that was to be reused for the Qt-based radio was also reviewed and found to be in a very poor shape. It had no documentation, and the code did not seem to follow any consistent conventions. Years of quick fixes and careless additions without refactoring had left their mark on the app engine.

At some point in the past, the radio stations of the old radio application had needed new data elements. However, rather than adding the data elements to the preset utility, all of its sources had been copied inside the app engine, and the modification had been done there. The app engine had swallowed the preset utility. This ghastly decision had been made mainly out of convenience. The Symbian middleware, along with the preset utility, had been maintained in Dallas, Texas. Communicating between time zones had been difficult, when organizing a simple phone call meant that the people in Oulu had to stay late at the office and the people in Dallas had to wake up early. This difficulty was finally removed by transferring the radio middleware maintenance responsibility to us in Oulu. This meant that the preset utility sources could be removed from the app engine, where they did not belong.

Because of other similar decisions where corners were cut, the app engine was in such a bad shape that alternatives needed to be considered. An already discontinued radio application called “Visual Radio” was coincidentally slightly familiar from previous employment. With some help from former colleagues, the source code was eventually found, and it seemed to be in a much better condition. The architecture was clear, and the necessary refactoring had been done. It was decided that the existing app engine would be discarded, and that the app engine from Visual Radio would be used instead. It had a very modular structure, which meant that many unnecessary features could be easily removed by simply discarding certain modules. Although the Visual Radio app engine was in a good shape, that project had had different requirements and goals than our project had, and refactoring was needed to realign its architecture to serve the new radio application better.
One thing that needed fixing was the fact that the app engine’s main API header file included headers from the radio middleware. This means that the middleware headers also get included in the application that includes the app engine header. This forms an unnecessary dependency between the UI and the middleware and breaks architectural layer separation. The Symbian layer separation is respected because the UI and the app engine both reside in the application layer. However, the new radio application was intended to have a similar layered internal structure internally, which meant that the app engine needed to hide the middleware from the UI. This problem was easily solved with the pImpl design pattern, and the new app engine header does not contain anything that the UI layer does not need [11 p. 15].
7. NEW FEATURES

The new radio application supports all of the features supported by the old Symbian-based radio application, which include the basic operations expected from a radio application. The user can scan the entire frequency band for radio stations, seek for them manually, edit station names, and obviously listen to radio broadcasts.

In addition to the existing features, the new radio had to support a number of new ones. Some of the new features were known from the beginning of the project, and some were added as the work progressed. The aim was nonetheless to keep the amount of new features, and therefore the amount of development time, at a reasonable level in order to remain within the tight project schedule.

7.1. Automatic region selection

One requirement for the radio application was the ability to choose the radio region automatically instead of asking the user for it, like the old radio had done. The manual region selection was seen as an annoying question that many users would not understand.

The Frequency Modulation (FM) broadcast band varies between different parts of the world. The frequency bands are shown in Table 1. The peculiar 50 kHz frequency spacing used in the Italian region is a relic from the past. Other European countries discontinued the 50 kHz frequency spacing after an International Telecommunication Union (ITU) conference decision in 1984 [13]. While supporting the Italian frequency spacing was not seen as a mandatory requirement, the support was still kept because the old radio had had it.

<table>
<thead>
<tr>
<th>Countries</th>
<th>ITU region</th>
<th>Band (MHz)</th>
<th>Spacing (kHz)</th>
<th>Possible stations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Europe, Africa</td>
<td>1</td>
<td>87.5 – 108.0</td>
<td>100</td>
<td>205</td>
</tr>
<tr>
<td>Italy</td>
<td>1</td>
<td>87.5 – 108.0</td>
<td>50</td>
<td>410</td>
</tr>
<tr>
<td>U.S.A.¹</td>
<td>2</td>
<td>88.1 – 107.9</td>
<td>200</td>
<td>99</td>
</tr>
<tr>
<td>Japan</td>
<td>3</td>
<td>76.0 – 90.0</td>
<td>100</td>
<td>140</td>
</tr>
</tbody>
</table>

The automatic region selection attempts to get the device location by checking the Cell ID, which is an identifier for the global system for mobile communications (GSM) base station [14]. The user is asked to enter the region only if the Subscriber Identity Module (SIM) card is not inserted in the device.

Supporting various frequency bands also poses severe legal concerns that must be taken into account. As seen in Table 1 the Japanese FM radio frequency band reaches

¹ The exact values of the American FM band vary depending on the sources but the values shown are used in Symbian.
very low, and it in fact overlaps with military or police frequencies in some countries. Device manufacturers must therefore disable the support for Japanese region in variants that are sold in countries where those frequencies are not allowed. For this reason, the radio application has to have a variation flag that can be used to disable the Japanese region setting.

7.2. The frequency strip

One of the first UI design ideas was a horizontal strip that was modeled after old car radios. In a car radio the user can tune to different stations by moving a vertical red bar along the frequency band. In the radio application the vertical red bar stays in the middle and the frequency band scrolls and loops round. Figure 11 shows the early prototype of the frequency strip.

The frequency strip posed rather heavy performance requirements for the engine. The frequency strip was intended to behave as a weighted wheel. The user can flick it left or right, and it should initially scroll fast and then slowly come to a stop. This provides a nice UI feature and an intuitive way to quickly find stations at the far ends of the frequency band.

For the engine, this kind of fast scrolling meant hundreds of tune requests in a short amount of time. As seen in Table 1, the 50 kHz frequency spacing in Italy, the worst-case scenario for the engine, amounts to 410 possible radio stations, and the user could easily flick through all of them in a short amount of time.

The tune request has to go from the UI through the UI engine, app engine, radio server, and radio adaptation server before finally reaching the receiver chip that takes a few milliseconds to tune to the frequency. After this, the tune notification has to trickle all the way up to the UI to complete the tuning operation. The exact time it took to perform a single tune operation was never measured. It was recognized to be clearly impossible and unnecessary to tune to all of the frequencies on the way to the final desired frequency.

Multiple solutions were prototyped to solve this, and the idea of running the engine in a separate thread was a constant consideration. In the end, the simplest solution of performing the tuning after a short delay timer was chosen. When the frequency strip is scrolled quickly, all of the tuning operations simply set the desired frequency and restart the tuning timer, effectively ignoring the frequencies that are passed by quickly. When the user scrolls more slowly, the timer has time to fire, and the frequency is tuned. The exact delay for the timer was a constant topic of
discussion, and the value was fine-tuned when new hardware builds were taken into use.

Figure 12 shows a simplified sequence that is executed when the frequency strip moves enough to cause tuning to another frequency. The figure uses the same color-coding that was used in Figure 5. The top part shows the sequence of tuning to a new frequency from the frequency strip, and the bottom part shows the sequence of notifying the frequency strip about the changed frequency. The frequency change may not originate from the frequency strip, for example if the user changes the station from headset buttons. In this case, only the bottom sequence is performed, and the frequency strip must update itself to reflect the updated frequency. On the other hand, the frequency strip must ignore all frequency change notifications that are responses to its own requests because it may have already been scrolled to yet another frequency. Therefore all frequency change requests are accompanied by a “change reason”, which is reported back in the frequency change notification. This allows UI controls to ignore frequency change updates that they themselves have initiated.
7.3. Favorite radio stations

One of the first new feature requests identified by the Product Owner was the need for favorite radio stations. The user had to be able to tag a radio station as a favorite, and it had to be then displayed more prominently in the UI. The user had to have an intuitive way of switching between favorite radio stations while the non-favorite ones had to be available but not in the way.

The implementation of tagging stations as favorites was trivial, but the UI design proved to be very difficult. There seemed to be no natural way to transition from having no favorites to having multiple favorites and still keep the UI behavior predictable and intuitive. While the UI design is not in scope of this thesis, it is simply noted that many different UI design concepts required modifications to the design. In some concepts, the user was able to loop through favorite stations as well as non-favorite ones, requiring the engine to have two separate station models to support both lists. In the end, an acceptable compromise between usability and functionality was reached, and the support for favorite stations was kept.

7.4. RDS support

RDS is a communication protocol for embedding small amounts of digital information in conventional FM radio broadcasts. RDS standardizes several types of information to be transmitted, and the radio application, like its predecessor, had to support some of them. The available RDS information fields and the level of support for them are summarized in Table 2.

<table>
<thead>
<tr>
<th>Name</th>
<th>FM Radio support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative Frequencies (AF)</td>
<td>Support removed.</td>
</tr>
<tr>
<td>Clock time (CT)</td>
<td>-</td>
</tr>
<tr>
<td>Enhanced Other Networks (EON)</td>
<td>-</td>
</tr>
<tr>
<td>Programme Identification (PI)</td>
<td>Used as the station identifier.</td>
</tr>
<tr>
<td>Programme Service (PS)</td>
<td>Used as the default station name.</td>
</tr>
<tr>
<td>Programme Type (PTY)</td>
<td>Stored, not yet used.</td>
</tr>
<tr>
<td>Regional (REG)</td>
<td>-</td>
</tr>
<tr>
<td>Radio Text (RT)</td>
<td>Shown to the user in main view.</td>
</tr>
<tr>
<td>Traffic Announcement (TA)</td>
<td>-</td>
</tr>
<tr>
<td>Traffic Programme (TP)</td>
<td>-</td>
</tr>
<tr>
<td>Traffic Message Channel (TMC)</td>
<td>-</td>
</tr>
</tbody>
</table>
7.4.1. **AF field support**

Alternative Frequencies (AF) is a field that radio stations can use to send a list of alternative frequencies that the station may use. The purpose of the list is to offer an opportunity to re-tune to a different frequency when the signal becomes too weak, e.g. when moving out of range. This is often utilized in car stereo systems without the listener even realizing that a frequency jump has been made.

The Symbian-based radio supported AF, so initially it was also implemented in the new radio. However, field-testing done by driving from Oulu to Haaparanta and around Munich showed the feature to be very problematic in a mobile device with only a single receiver chip.

Car stereo systems typically have two receivers, leaving the second receiver to scan the list of alternative frequencies determining if a frequency jump would be favorable. If it deems the jump to be favorable, the receivers can simply switch roles without the listener noticing it. The second receiver takes over playing the radio broadcast, and the first one assumes the role of scanning through the alternative frequencies.

The fact that a mobile device has only one receiver means that the user has to listen to corrupted radio broadcast for quite a while before the application determines the signal to be too weak and attempts to jump. Moreover, since the device has only one receiver, the radio application has no way of knowing which alternative frequency, if any, has a stronger signal. These facts mean that the jump is always performed too late and blindfolded, and success is only based on luck. Due to these reasons, the support for AF was dropped from both radios, old and new.

7.4.2. **PI field support**

Programme Identification (PI) was used since the beginning of the project as the unique station identifier. It was used when comparing the equality of radio stations to make sure the correct station was updated when RDS data was received.

PI was also used when scanning the entire frequency band. When the user scans the frequency band in another city, the result is likely to be a different list of frequencies that had previously been saved. However, the list may still contain some of the same radio stations, only broadcasting from a different frequency. In this case, the PI code could, if received, be used to confidently identify the radio station and update the new frequency into the saved station. However, if the signal strength is poor and PI is not received, a confident identification cannot be made and the frequency has to be saved as a new station. The original station, and any data contained within it, would then be removed.

In the American region, the PI code was used to calculate the station call sign. The stations in America typically do not use human-understandable station names sent through the PS name field described in the next chapter. Instead, they use a concoction of letters called the call sign, which can be calculated from the PI code. The call sign for a station in New York, for example, is WNOW-FM.
7.4.3. **PS field support**

Like the PI code, the Programme Service (PS) name was used since the beginning of the project. Radio stations can use the PS name field to broadcast their name, for example “Nova” or “YleX”. The radio application stores the PS name as the default station name and shows it to the user.

Some radio broadcasters have decided to misuse the PS field and send advertisements through it. The PS name field is only 8 characters long, so the advertisements are cut into 8-character pieces that are sent one after the other. The interval between the parts varies between stations. This type of misuse of the PS field was termed “Dynamic PS”.

To the radio receiver, this appears as a station that changes its name constantly. This is not allowed by the RDS specification, but it is not strictly forbidden either, so the practice is not rare. Radio Mega was the only radio station in Oulu to do this. The problem arises when the radio station name is saved in the radio application. The actual radio station name cannot be saved because the station does not send it. Instead, an unsophisticated implementation would save a part of the advertisement that happened to be broadcasted at the time when the station was saved. Saving a random part of an advertisement as the station name was unacceptable, as was changing the station name every couple of seconds. The thought of the station list flickering when multiple stations constantly change their names was considered to be confusing and annoying to the user.

It was decided that the radio should attempt to detect if a station uses Dynamic PS, and if it does, simply ignore any PS data that the station sends. A small delay timer was added to the reception of PS name just before it is shown to the user. If the station does not send another differing PS name within that period, the station is considered to be conforming to the RDS standard, and only then is the name saved.

7.5. **RT+ support**

The support for Radio Text Plus (RT+) was a new requirement for the new radio. It was not supported by the old radio, and the drivers and adaption had to be updated to send the RT+ tags to the radio client.

RT+ is an extension to the RDS Radio Text (RT) standard [15]. The normal RT consists of 64 characters that are sent from the radio station, and RT+ uses some of those characters as special tags that can be used to point out an artist and a song name from the text, for example. One block of characters can contain 2 tags, and if only one is used, a special empty tag is used for the other one. There are over 60 possible RT+ tags, and the radio supports some of them. Table 3 shows some of the most interesting tags and their level of support in the radio application.
<table>
<thead>
<tr>
<th>Tag name</th>
<th>Description</th>
<th>Support in radio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItemTitle</td>
<td>Song name</td>
<td>Saved to song history</td>
</tr>
<tr>
<td>ItemArtist</td>
<td>Song artist</td>
<td>Saved to song history</td>
</tr>
<tr>
<td>ProgramHomepage</td>
<td>Radio program home page</td>
<td>Saved to RadioStation</td>
</tr>
<tr>
<td>ItemGenre</td>
<td>Radio station genre</td>
<td>Support planned</td>
</tr>
<tr>
<td>InfoURL</td>
<td>Information URL</td>
<td>Support planned</td>
</tr>
<tr>
<td>StationNameShort</td>
<td>Short station name</td>
<td>Support planned</td>
</tr>
<tr>
<td>StationNameLong</td>
<td>Long station name</td>
<td>Support planned</td>
</tr>
<tr>
<td>PhoneHotline</td>
<td>Hotline phone number</td>
<td>Support considered</td>
</tr>
<tr>
<td>PhoneStudio</td>
<td>Studio phone number</td>
<td>Support planned</td>
</tr>
<tr>
<td>PhoneOther</td>
<td>Other phone number</td>
<td>Support considered</td>
</tr>
<tr>
<td>SmsStudio</td>
<td>Studio SMS number</td>
<td>Support considered</td>
</tr>
<tr>
<td>SmsOther</td>
<td>Other SMS number</td>
<td>Support considered</td>
</tr>
<tr>
<td>EmailHotline</td>
<td>Hotmail email address</td>
<td>Support considered</td>
</tr>
<tr>
<td>EmailStudio</td>
<td>Studio email address</td>
<td>Support considered</td>
</tr>
<tr>
<td>EmailOther</td>
<td>Other email address</td>
<td>Support considered</td>
</tr>
<tr>
<td>MmsOther</td>
<td>Other MMS number</td>
<td>Support considered</td>
</tr>
</tbody>
</table>

The support for the RT+ tags was added as a separate callback in the radio hardware observer interface instead of modifying the existing RT callback. The normal RT callback is called first with the untagged content, and the RT+ tag callback is called a moment later. What is even worse, the tags come with separate callbacks one at a time, and they may arrive in any order. These unfortunate design decisions meant that RadioStationModel needed special logic in order to properly receive the RT+ content. When the RT callback is called, a timer is started to wait for the RT+ tags that may possibly arrive soon after. If the timer expires and no tags have arrived, the RT is sent to the UI as it is. This is a normal situation when the content is simple text without any tags. If the RT+ tags do arrive before the timer expires, they are identified and appropriate action is taken to handle them. If the RT+ tags specify an artist and a song title for example, they are sent to RadioHistoryModel to be saved in the song history database.

7.6. Song history

One of the new features that build upon the RT+ support is storing a list of songs that were played while the user was listening to a radio station. The artist and the song title may have been received as RT+ tags in the RT, or the user may have used a third party application to recognize the song.

The user is able to view and edit the song history list and purchase songs through the music store. The design of the song history feature brings up no new design considerations, so the details are omitted from this thesis.


8. IMPLEMENTATION

The implementation started when the architectural designs had been approved. The implementation followed the design principles and best practices outlined in the bachelor’s thesis [11]. The principles were discussed in the team whenever issues came up in daily meetings and were enforced in the code reviews. Sticking to the principles and guidelines sometimes caused heated discussion, but no major exceptions were made. This helped to ensure that the implementation was clear, understandable, and robust.

The entire code base including the reused Symbian-based modules was reviewed once, and the Qt-based modules were reviewed again later. The reviews were held informally and internally in the team. Findings were discussed, and corrections were added to the backlog and implemented at the earliest convenience. In addition to the full code reviews, desk checks were done to verify new features and other modifications to the code.

The implementation of UI concept prototypes had begun much earlier than the final application implementation. Initially the prototypes were used as test beds for the new UI controls, but eventually the prototypes were discarded. The final application is not based on a prototype and does not reuse code from one. The UI controls that were developed on prototypes were thoroughly reviewed and refactored before being adopted into the final application. [11 p. 25]

8.1. Discarded solution proposals

Many ideas and design considerations were evaluated as the project progressed. Many of them were dismissed quickly, while some of them were discussed and studied carefully. Some even required testing with a proof-of-concept prototype.

The ideas described in this chapter were ultimately abandoned and never made it into the final design. While they have their merits, they were considered not to improve the design and in some cases even to harm it.

8.1.1. Threading

Threading was considered on a number of occasions as a possible solution to a problem caused by one of the UI controls. As described in Chapter 7.2, the frequency strip that presented a very pleasing UI experience also posed a serious performance problem. With one flick the user could span dozens of frequencies, and the engine simply could not keep up if it attempted to tune to all of them.

Placing the engine in its own thread was one possibility of dealing with the performance problem. The communication between the UI thread and the engine thread would have been done by signals and slots, which would have prevented any synchronization issues. A proof-of-concept prototype was implemented and tested,
and while it largely solved the performance problem, it also complicated the design. In the end, when a balance had to be struck between threading and an alternative solution using a simple timer, the simplicity of the timer weighed more.

8.1.2. DLL reduction

Near the end of the project when memory usage was being monitored more carefully, the option of compiling the DLLs into the executable was studied. The use of Qt in Symbian causes a substantial amount of memory waste. Qt uses Writable Static Data (WSD), which has to be enabled for every DLL that uses Qt [11 p. 29]. This means that 4 kilobytes of memory has to be allocated for each DLL to store the WSD variables.

With project file flags, it is possible to differentiate the compilation process so that the DLLs are compiled into the executable in the Symbian builds. The unit testing still required them to be compiled as DLLs that could be easily linked into the test modules. The reduction was ultimately not done, but it was left as a future improvement possibility.

8.1.3. Parallel startup with radio server

It was noticed that the radio did not meet the strict startup time requirements set for all new applications. Quick timestamp logging showed that a lot of time was spent initializing the radio receiver chip. A patch firmware needed to be loaded into the chip before it could be used. There was no way to speed up the hardware initialization, but the software could be optimized to begin the hardware initialization sooner in the startup process.

Figure 13 illustrates the differences between sequential and parallel startup sequences. Time flows from left to right, and the arrows signify components that are being created. Green color is used to highlight components in the radio application process, while the radio server runs in its own process and the radio hardware driver runs in the radio adaptation process.

In the sequential startup sequence, the components are created one after the other, and the radio server process is started when the radio server client attempts to connect to it and discovers that it is not running. The radio server is, in Symbian terminology, a “transient server”, which means that it shuts itself down when there are no clients using it. When the radio application is closed and restarted within the radio server shutdown timer period, the startup is fast because the server is already running. In the typical startup scenario, however, the server is not running and the startup is slower.

The parallel startup sequence was designed to optimize the startup time by kick-starting the radio server process as early as possible in the startup procedure. Platform-independence of the engine is not compromised because the engine does
not attempt to communicate with the Symbian server. It simply launches the server executable without arguments and then forgets about it. The server starts initializing itself in parallel with the radio application, and when the radio server client is constructed, it simply connects to the already running server. A proof-of-concept implementation was done and found to work as required. To avoid regression near the release date, the parallel startup sequence was ultimately not taken into use.

Figure 13. Sequential and parallel startup sequences.

8.2. Temporary solutions

Ideal designs that fulfill the requirements beautifully can sometimes be squashed by real world limitations on what can actually be accomplished within the project schedule. Nokia had only recently begun porting Qt to Symbian when the radio design started and all facets of the Symbian platform were not ready to be used through Qt. However, this posed merely an idealistic problem where everything was still possible, only not quite in the optimal way.

Solutions had to be found though, so a set of temporary solutions was devised. Their temporary nature was documented and they were set on the project backlog to be refactored later.
8.2.1. Radio settings

Qt contains a class called QSettings, which is used for storing various persistent settings needed by an application. For the radio application, this would have been the natural place to keep the last tuned frequency, selected region, volume level, and other similar settings.

Unfortunately, the QSettings was not available yet, so another solution had to be found. Symbian keeps application settings in Central Repository. It was known that QSettings would eventually be ported to use Central Repository, so the simplest temporary solution was to use it directly, and to switch to QSettings later when it would become available. The reused app engine already had a settings implementation, which only needed to be exposed through the wrapper API.

8.2.2. Media key handling

The user is able to control the radio playback and the volume level with the buttons in the headset. The engine would have been the most logical place to handle these media key events. As with the settings, however, the media key observing was not yet available through the Qt Mobility API.

Similarly, the reused app engine already contained a Symbian-based implementation to monitor the media keys, which was exposed through the wrapper API. The resulting solution is not ideal, as it involves an unnecessarily long function call chain to respond to the key events. The event is caught in the app engine at the lower level of the application, and it is sent upwards to the UI engine that decides how to act upon it and then sends a command back down to the app engine.

8.3. Memory management

Memory management was a critically important design consideration in the entire application. As with all mobile applications, it is crucial to ensure that no memory is leaked in any situation. This means designing the architecture and code in such a way that destroying objects does not rely on the carefulness of the developer but is handled automatically by the C++ compiler.

Complying with the “Resource Acquisition Is Initialization” (RAII) principle, all allocated resources are owned by smart pointers except when the parent mechanism can be used. Every class and function is designed to expect an exception to be thrown at any time.
8.3.1. **Object lifetime management**

The lifetime of the Qt-based objects is managed by setting the parent pointer when a suitable parent exists. When a suitable parent is not available or the class does not support parents, the Qt smart pointers are used. Normal C++ pointers are never used for owned data, and the code contains no manual `delete` calls. [11 p. 27]

The implicitly shared `RadioStation` instances are a special case with regards to object lifetime management. The implicit sharing ensures that the `RadioStation` instances are safe to pass around by value without worrying about lifetime management. The compiler is responsible for freeing the passed `RadioStation` instance, which in turn is responsible for freeing the shared data instance and uses reference counting to do it.

8.3.2. **Smart pointers**

The entire radio application uses smart pointers for all owned data, except when Qt parent-child mechanism can be used [11 p. 26]. Normal C++ pointers are used only when the target object is not owned and therefore does not need to be freed. The C++ pointers are always set to zero when they are no longer used.

When interfacing with Symbian code in the wrapper, smart pointers are used exclusively to own instances of Symbian objects in Qt classes. Qt smart pointers are also used when Symbian classes in the wrapper own other Symbian classes. The general principle is to keep the Symbian-related code and Qt-related code as separate as possible, but the use of smart pointers removes the possibility of forgetting to call `delete` in the class destructor. The safety of never accidentally creating a memory leak overruled the general principle in this case.

Weak pointers are a type of smart pointer where the pointer does not own its target and therefore does not destroy it. Its purpose is to observe the lifetime of the target object and set itself to zero when it notices that the target object has been destroyed. Weak pointers are used in some situations where the engine is responsible for creating an instance of a class, but the ownership is transferred to the caller and the engine needs a pointer to the instance.

8.4. **Competitor comparison**

The radio application was constantly compared against competing applications on Android, iPhone, S40, and Windows Phone devices. Any disadvantages were carefully investigated, and improvements were made when possible.

One situation where an improvement could not be made was the scanning speed. It was noted during a testing session that scanning for available stations progressed much quicker on a Windows Phone than on the Symbian device. The `RadioScanner`
class was optimized for maximum speed, but the goal of matching the scanning time of the Windows Phone could not be met. Coincidentally, a new prototype device family was slowly being taken into development, and it was tested to perform as fast the Windows Phone. The faster processor and a different radio receiver chip made all the difference.

The favorite station concept was also compared to those of the competitors, and it seemed that none of them had a good solution to the slight usability inconsistency that the favorite stations posed. The inconsistency arises when the user has no favorites and skips through stations one by one. In this case, the skip operation traverses the list of all scanned stations. If the user does have favorites, the skip operation should traverse the list of favorites because the user has indicated that they are more interesting than the other stations. The problem comes from the transition phase between traversing the list of all stations to the list of favorite stations. Ideally the transition should be made when the user adds the first favorite station, but skipping through a list of one does not make sense. The application would seem unresponsive if pressing the skip button did nothing. The transition can be made when the user adds a second favorite station. Most of the competitors had solved this issue by not supporting favorite stations at all. The ones that did support them also had the same inconsistency.

8.5. Problems

As any research and development project, this one was not without problems. The tight schedule was a constant source of problems by requiring quick solutions to complicated problems.

Although the problems seemed severe at times, all of them were eventually solved, or at least a reasonable workaround was devised. There were too many problems to count them all, so only the most relevant ones are described.

8.5.1. The problem with workarounds

Although often necessary, the workarounds pose a problem by themselves. The Orbit framework, for example was a constant source of headaches, as it was still being developed when the radio application was being built on top of it. This meant that many of the services it offered worked incorrectly or started to misbehave after having worked perfectly in the previous release.

Often the problem could be relatively easily fixed with a temporary workaround. However, great care needed to be exercised on implementing the workaround in such a way that it could be easily removed later and on documenting the workaround in high detail. A workaround that is not removed when it becomes unnecessary becomes a source of bugs in its own right.
8.5.2. C++ templates with Qt

The development of the RadioStationModel hit a serious problem when it tried to combine C++ templates with signals and slots. The model class used templates to generalize its usage, and it also attempted to template a slot. The code did not work as expected, and after a lot of debugging and googling, it was discovered that templates and the Meta-Object Compiler (MOC) do not work together. [11 p. 17, 16]

There was no solution or workaround to be found this time. The only working solution was to abandon the template design and to re-implement a new one without templates. In hindsight, the new design is much simpler and therefore easier for everyone to understand.

8.5.3. The constantly changing platform

One of the biggest sources of problems and frustration was the fact that the platform was still heavily in development while the applications were being developed on top of it. Qt had only recently been ported to Symbian, and it was not fully supported yet. The most critical features had been ported, but some features were still under development. On the other hand, Qt was still considerably more stable than the Orbit UI framework that was still far from being completed. Orbit still underwent major changes during its development cycle and thus required all of the applications to be updated in order for them to work.

One particularly painful situation occurred when the Orbit team had decided to deprecate a base class called HbAbstractStrip that they deemed unnecessary. A quick search through the platform sources had revealed that the class did not have enough users to justify its existence. In fact, the radio application was the only one using it as the basis of the frequency strip. The reason for using it came from the fact that HbAbstractStrip supported looping round, which was a key requirement for the frequency strip. After HbAbstractStrip had been deprecated, a study was conducted to determine which Orbit base class could replace it. It was clear from the beginning that no other base class could be easily adopted, and a thorough redesign of the frequency strip was required. It was found that HbGridView was the only alternative that supported looping and could be used to build the frequency strip. It was taken into use, but near completion, it was noticed that HbGridView varied the spacing between elements that are scrolled. For the frequency strip, it was absolutely unacceptable that the spacing between frequency markings would not be the same everywhere. The almost complete implementation had to be discarded, and eventually the sources from HbAbstractStrip were adopted into the radio after its unnecessary features were removed.

Even worse than removing critical base classes was changing the behavior of the API functions in subtle ways that sometimes caused devastating problems. The deprecation of a base class was always preceded by a notification of the removal so that teams had time to prepare. The subtle change in the API function behavior might have only been accompanied by a vague one-liner in the release note or maybe nothing at all. The problem was noticed only when the application started to behave
incorrectly. The root cause was sometimes difficult to track down and required thorough debugging. Because of situations like these, a common practice of always testing with the previous Orbit release as well was adopted. In this way, it was possible to check if the error had suddenly appeared with the new framework release.

8.5.4. Time for refactoring

Finding the time and getting the management’s permission for refactoring was sometimes a problem. Refactoring does not produce any new features and does not always even fix errors directly. It is therefore easy to dismiss and postpone until the “more important” things have been completed.

Software development is an iterative process and requires constant re-evaluation of previous design decisions. Especially when the underlying platform keeps changing, it is crucial that the necessary time for refactoring can be allocated. The immediate management in the team was well aware of this, but convincing the upper management was sometimes tricky.

The amount of time taken by refactoring can be kept to a minimum by performing the refactoring as soon as the need arises. This ensures that all subsequent development is done after the refactoring has been done and the system is again in an optimal state to logically accommodate the new code. This point was highlighted when the refactoring of RadioStation handling inside RadioStationModel was postponed. This resulted in the RDS reception logic being implemented twice: once before and once after the refactoring. [11 p. 22]

8.6. Testing

The goals for testability and simplicity support and complement each other nicely. Writing automated tests for simple components is easy, and good testing helps to ensure the component correctness and to keep it simple. The goal of simplicity was notably achieved when test cases could be easily written without consulting the author of the component.

Throughout the project, every released version was put through multiple types of manual testing rounds. Smaller test sets were run almost daily, while the bigger sets were run for each release.

Separate unit tests were implemented to test the engine and other components individually. As the engine was a Qt-based component, it could be tested with QTestLib, while the Symbian-based components had to be tested with a Symbian-based unit-testing framework.

No special tests were done to test for dependence on Avkon or Symbian OS. This test happened automatically when software builds without Avkon were taken into use and also during development in the Windows workstation. The former testified for
the lack of Avkon dependency while the latter also testified for platform-independence.

Automated testing was done with the “Matti” unit test framework by writing Ruby scripts to perform the test actions. The tests operated the application by generating touch events to the UI and by observing the output. The tests therefore tested the entire application, not just the engine. The goal had been to automate the testing as much as possible, but manual testing was also necessary for some use cases. Testing the audio output and headset connection was always a problem for the automated tests. Near the end of the project, a special headset had been developed that allowed the computer running the tests to “listen” to the audio. In practice, this meant that the test script was able to check if something was being sent through the audio jack.

Memory consumption was monitored throughout the project by testing the application’s main use cases with a special memory-monitoring tool. The tests were run repeatedly to find any leaks. The leaks that were found were reported and fixed. The traces were compared with previous runs to see how the overall memory consumption of the application had changed. One of the earlier versions of the frequency strip, for example, was found to consume too much memory and was subsequently redesigned in order to minimize its memory consumption.

Special testing sessions were arranged in addition to the normal testing process. The sessions included people from all teams, and the idea was to test unfamiliar applications as an end-user. The Product Owner observed the testing carefully, recording even the smallest comments, and asked questions about the first impressions of using the application. The resulting test report was reviewed, and any possible problems in the user experience were addressed.

Another type of special testing session was unofficially called “torture testing”. The team members attempted to break the application forcefully by quickly tapping on the screen and generally doing anything weird they could think of. A special radio signal generator device was used to send broken RDS data to see if it would be handled incorrectly. The engine fared well in the torture tests and did not need any major changes to the design. The most significant change was the refinement of the tuning timer that allows the engine to ignore tuning requests that arrive in rapid succession from the frequency strip.
9. DISCUSSION

9.1. Results

The design and implementation of the radio application succeeded surprisingly well considering the complicated starting point. Simplicity from complicated beginnings is a difficult thing to achieve, but through repetitive refactoring and gradual improvements it was eventually achieved. The radio was one of the first applications to reach the code-complete milestone, which meant that the implementation of the main features was completed.

The project requirements were fulfilled, and the goals were met. All of the required features were supported except for AF that was also removed from the Symbian-based radio. The new features were successfully implemented and tested to work as required. The goals of platform and UI independence for the engine were met. The refactoring and reuse of the existing radio modules succeeded surprisingly well after an already buried engine from Visual Radio was revitalized. Memory consumption as well as startup time stayed within reasonable levels, although achieving both goals required quite a bit of work. It was considered that the requirements for simplicity and readability had been achieved when an additional developer was temporarily assigned to the radio team and was immediately able to understand the design and produce valuable code.

The decision to reject the requirement of using the multimedia application framework proved to be the right one. A neighboring team had implemented their application by using the framework, and they struggled to cope with its tight restrictions. As a result of using the framework, their application is not platform-independent as the framework exists only in Symbian OS. The radio application is much better off without it.

The high-speed camera was used extensively to optimize the startup sequence of the UI, but the target of a 900-millisecond startup was not quite met. In the end, the startup took over a second, but it was still considered acceptable. The flickering had been cleared out, and the startup looked smooth.

Although the end result was good, mistakes were made on the way. Mistakes that cost time that could have been used more productively. The design went through unnecessary revisions when wild ideas were attempted. The goal of simplicity was sometimes forgotten during the development, which resulted in attempting to use C++ templates with Qt slots and in several other bad ideas. Some of the many revisions that certain components went through were not all due to the changing platform or project requirements. Some were caused by foolhardy attempts to try a new technique while producing production code. Trips to Qt Developer Days in Munich in 2008 and 2010 taught a lot and inspired several new implementation ideas that sometimes sacrificed simplicity [17]. Eventually reason prevailed, and the code was refactored with simplicity in mind.

One thoughtless mistake was to implement the wrappers as separate modules. The “Radio Preset Storage” module is extremely simple, and there is no reason why it should be a separate DLL instead of being just another service offered by the engine wrapper. Merging it into the wrapper would reduce the amount of DLLs, thus
reducing the memory usage. The app engine and preset utility reside in different layers in the Symbian OS, which was the original motivation for keeping the wrappers separate. However, as the wrappers themselves reside in the same layer, the separation was a mistake that should be corrected.

9.2. Reviews

The architecture review board reviewed the designs twice. The first review was held before the implementation was started, and the second was held when most of the features had been completed. The second review also verified that the implementation had been done properly and had followed the design and general coding guidelines.

The reviews began when the Package Owners of each application posted the designs and sources to a wiki page. After that, the other Package Owners and the architecture review board reviewed the designs. The comments were discussed in a review meeting where the Package Owners presented and defended their designs. In both reviews, the radio design and implementation received good comments and passed without “actions”. This meant that the design was approved and nothing needed to be corrected.

A “Kudos” performance award was given for the design work. It was noted that the radio was one of the few applications that contained absolutely no Symbian code above the wrappers and had therefore met the requirement of platform-independence. The notification email about the Kudos award is shown in Appendix 4.

9.3. Windows version

Developing the application for two platforms at the same time from a single code-branch proved to be a good strategy, and the extra effort needed to implement the Windows version was well worth it. In addition to ensuring that the design and implementation stays platform-independent, the strategy also had some unexpected benefits.

Although the process of porting Qt to Symbian had progressed far enough that it could be taken into use, it was still not complete when the implementation of the radio application was started. Corners had been cut in order to speed up the project, and the Qt-port contained quick temporary solutions that depended on the old Avkon UI library in a few non-critical areas. The dependencies on Avkon were eventually removed, but it meant that the development environment included the Avkon library for quite a long time. This introduced a risk of carelessly developing an application that had an unintentional dependency on Avkon. Such applications were broken when Avkon was removed from the environment. Developing the application also for Windows made sure that no accidental dependencies on Avkon were formed and its removal from the environment had no effect on the radio application.
The actual Windows version proved to be very useful also in other, surprising ways. When UI designers and various managers needed to see what the radio looked like and how it worked, they did not need to have a prototype phone flashed with a particular image to test the radio. A release build of the Windows version was maintained in a network drive, and a link to the executable was sent via email. By clicking on the link, anyone could launch the radio application in their own Windows workstation and test the basic use-cases. The stubbed Windows version proved to be an advantage that no other team had. Screenshots of the Windows version are shown in Appendix 3.

9.4. Difficulty

Designing a simple system is surprisingly hard, especially when looking at the finished diagrams later in perfect hindsight. It all looks so simple and easy to understand, but making it that way took many iterations and a lot of thought. Near the end of the project, it was calculated that five completely different versions of the frequency strip and four versions of RadioStationModel had been implemented. They were not mere UML diagrams, but fully working implementations that were used until they needed to be replaced.

Writing the actual implementation on the other hand was surprisingly easy. The quality of the Qt documentation meant that the work progressed quickly and effortlessly. Also the intuitiveness of the Qt APIs meant that often the documentation did not even need to be consulted, because the classes typically did what their name suggested and contained all the services that one would expect.

9.5. The end of Qt radio and Symbian

In 2010, the former Trolltech, now called the Qt software division in Nokia, announced a new UI description language called Qt Meta-Object Language (QML) and its supporting backend framework [11 p. 17]. QML had been developed with a small team of experts in a relatively small amount of time, and it far exceeded the capabilities of the more traditional Orbit framework.

On October 26th 2010, we were all taken by bus to Club Teatria in Oulu to hear about a strategy update. It had been decided that the development of the Orbit framework would be discontinued and that QML would replace it. All application teams had to remove any dependencies on Orbit and redesign a new UI with QML. The original decision of replacing Avkon with a completely new UI framework had also been reconsidered, and Timebox 10.1 was cancelled.

According to the new plan, only a selected set of applications were to be implemented with QML, and they were to coexist in the device with the old Avkon-based applications. FM Radio was one of the chosen ones, and the redesign started immediately. I had been debugging a problem in an Orbit component before stepping
on the bus, and the debugger was waiting for me when I got back. I closed the debugger and removed the Orbit dependency from the radio before going home. After a few days, the radio was running with a new QML-based UI, making it one of the first ones to do so. The UI was crude, but it supported the main features and demonstrated that the engine worked without Orbit.

Removing the Orbit dependency was very easy since the only part that depended on it was the UI. The engine required no changes, but later the API was slightly refactored to better serve the QML UI. During the phase of designing a proper UI with QML, many proposals and design concepts were implemented. Some were radically different from what the engine had initially been designed to support, but the engine worked without problems and continued to support a wide variety of differing UI ideas.

The plan of using the radio engine in Meego was rejected by a management decision, and the N9 eventually shipped without a radio. Keeping the engine clean of UI dependencies was initially motivated by the possibility to run it on Meego, but the decision proved to be a good one even though that possibility was never realized. The lack of dependency on Orbit in the engine made it easy for the radio to adapt to the change in project strategy.

QML enabled the UI designers to imagine things that had never been possible before, and new UI designs were being proposed daily. FM Radio remained in the list of chosen applications for a while but was eventually dropped because of resourcing issues. The subcontractors responsible for maintaining the old radio were being let go, and our team needed to take responsibility of it. The Qt-based radio was officially killed.

On February 11th 2011, the new CEO, Stephen Elop, announced the discontinuation of Symbian in the famous “burning platform” speech. Everybody working on Symbian, me included, was transferred to Accenture, and the last Symbian device to be sold was the 808 PureView that shipped on March 2nd 2012 [18, 19]. The Symbian saga had ended.

Fortunately, the Qt framework lives on. Nokia continued its development for a while until finally selling it to Digia. Digia is developing it further and, at the time of writing this thesis, just released an alpha version of Qt 5.1 [20, 21].

9.6. Future development

Because the project was cancelled, the implementation was never finished, but plans for future improvement already existed. Due to scheduling constraints, some corners had to be cut and the optimal implementation could not be achieved within the schedule of the initial release of the application. Detailed plans for future releases were formulated. The application was intended to be upgraded with IAD updates. New features would have been introduced gradually, and the application architecture would have been refactored to remove any temporary solutions.

It was clear from the start that reusing old Symbian-based modules was to be a temporary solution dictated by the tight schedule. A three-step migration plan was devised to re-implement the Symbian-based modules with Qt and to shift the wrapper layer down closer to the hardware adaptation layer. The IAD updates would
have been used to update the radio to become a fully Qt-based application. One aspect of this plan was the goal of developing a common radio application for both Symbian and Meego. Although it had not been done for N9, it was still seen as a future possibility.

QtMobility is an extension library implemented by Nokia to extend the standard Qt API to support common use cases for mobile devices. It contains features such as making a phone call, sending messages, and accessing the device location. The part that was relevant for us was the API to access radio features. Third party developers are able to implement their own radio applications by accessing the radio features through a class called QRadioTuner, which is a thin wrapper for the Symbian-based radio middleware [22]. It supports only the most basic features and lacks support for scanning for available frequencies. The new radio application did not use QtMobility because it was not available at the time of the implementation. It was recognized that in order to ensure good quality of the third party API for the radio, the radio application should be modified to use it. The plan was to move the radio UI engine behind the QRadioTuner interface and to extend its API to support the full set of radio features that the new engine offered. The radio UI would have been adapted to use the engine through QRadioTuner ensuring that it gets properly tested and maintained.

Digital Audio Broadcasting (DAB) is a new radio technology where the radio stations broadcast are sent digitally [23]. Digital broadcast has many advantages compared to the analogue FM broadcast. DAB can contain more radio stations over a specific spectrum, and it is more robust with regard to noise and fading. It can also offer a better audio quality and a much more comprehensive data service than RDS. Receiving DAB broadcasts requires a dedicated receiver chip that no device yet contained. Supporting DAB via a special DAB headset accessory was planned as an application update. The engine would not have needed any major changes, only some additional callbacks to receive the data services from the DAB receiver. The lower layers would have needed more elaborate changes to support the accessory.
10. CONCLUSIONS

The purpose of this thesis was to analyze a project to develop a replacement for the FM Radio application for the new Symbian device family. Some details of the project process and relevant events were discussed when they had an impact on the design work. The cancellation of the Orbit framework had an especially big impact on the design, as the entire UI had to be redesigned. The engine design proved to be robust enough to survive such a big change without requiring modifications. Later modifications were only made as a part of normal refactoring to better support the new UI.

The goal of the project was to offer a better radio user experience with multiple new features, the most notable being the support for favorite stations and RT+. The goal was achieved, although the usage experience was refined several times and completely redesigned after the Orbit cancellation.

The existing radio modules were refactored and reused as required, and the application would have been completed within the project schedule. The cancellation of the entire project means that the application will never be completed, but all of the features were implemented and the project was in target schedule.

The application was implemented with the Qt application framework, and its UI and engine are fully platform-independent. By the initial design, only the engine would have been platform-independent, and the UI would have been dependent on the Orbit framework as required. The cancellation of the Orbit framework and the subsequent redesign resulted in a fully platform-independent UI as well. The requirement to not have a dependency on Avkon was achieved, as witnessed by running the application on a device without Avkon and on a Windows workstation.

The development of the application on a Windows workstation helped to ensure platform-independence and made sure that no deviations from it were ever made.

The goals for testability and simplicity were achieved by supporting each other. Writing test cases for simple components was straightforward and required no additional documentation or consultation from the author of the component. Proper and easily executed test cases, on the other hand, helped to ensure the component correctness and to keep it simple.

The goal of sticking to established design patterns and not designing anything overly clever was achieved after some iteration. Some minor deviations were made, for example when trying to use C++ templates with Qt slots, but further investigation and refactoring removed such discrepancies. The resulting high-level design, as well as the detailed designs, is simple enough to be printed on an A3-sized poster and to be easily understood by all. The design and implementation stuck to the design principles and best practices defined in the bachelor’s thesis by the same author. The principles were discussed in the team and enforced in code reviews. No deviations were made, which resulted in a clear and robust implementation. The principles were found to be valid and necessary for the successful completion of a complex software project.

The changing platform added its challenges when the Qt framework was still finalizing its Symbian port and the Orbit framework was still heavily under development. Orbit in particular caused plenty of trouble by deprecating critical classes and subtly modifying API function behavior. The challenges were met and the problems were solved or a temporary workaround was found. Great care was
taken to ensure that the workaround indeed was temporary and did not accidentally become a permanent addition and a source of errors in the code.

The discussion in this thesis focused on the design of the engine module that housed most of the application logic and was therefore the application’s most important component. The wrapper that ensured the application’s platform-independence was also discussed in detail. The application UI was discussed briefly as it made its requirements on the engine design. The refactoring and reuse of the existing Symbian-based modules was also discussed briefly as they were relevant for the engine design. The redesign of the preset utility was covered because it was directly involved with saving the radio stations.

Finally it was discovered that designing a simple system from complicated beginnings is a difficult task. Formulating an optimal design and implementation requires many iterations and gradual refinements. Refactoring that is done early and often was found to be a key element in a successful completion of a software project.

The cancellation of the large-scale project does not diminish the success of the application project. It does, however, reduce its significance to the extent of this thesis. The application will never be seen by anyone, which is sad on a personal level. If the project had not been cancelled, it could have been a significant step towards a better user experience and a better application ecosystem.
11. REFERENCES


12. APPENDICES

Appendix 1. Foreword of the file logger article in Nokia Developer Wiki.

Appendix 2. RadioStation declaration in radiostation.h

Appendix 3. Images of the radio application running in Windows.

Appendix 4. The Kudos award received for the engine design work.
Appendix 1. Foreword of the file logger article in Nokia Developer Wiki.

Figure 14. Foreword of the logger article in Nokia Developer Wiki.
Appendix 2.  RadioStation declaration in radiostation.h

/*
 * Copyright (c) 2009 Nokia Corporation and/or its subsidiary(-ies).
 * All rights reserved.
 * This software, including documentation is protected by copyright controlled
 * by Nokia Corporation. Copying, including reproducing, storing, adapting or
 * translating, any or all of this material can only be done in accordance
 * with the Nokia Symbian License version 1.0 (or any subsequent update) or
 * any other license terms expressly agreed between you and Nokia.
 * This material contains Nokia's confidential information which
 * may not be disclosed to others without the prior written consent of Nokia.
 *
 * Initial Contributors:
 * Nokia Corporation - initial contribution.
 * Contributors:
 * Description:
 */

#ifndef RADIOSTATION_H_
#define RADIOSTATION_H_

// System includes
#include <QObject>
#include <QSharedDataPointer>
#include <QString>
#include <QMetaType>

// User includes
#include "radiouiengineexport.h"
#include "radio_global.h"

// Constants

// Class declarations
class RadioStationPrivate;

// Class declaration

class UI_ENGINE_DLL_EXPORT RadioStation : public QObject {
    friend class RadioStationModel;
    friend class RadioStationModelPrivate;
    friend class TestRadioUiEngine;
    friend class TestRadioPresetStorage;
    friend class RadioStationQml;
public:
/**
 * Flags to indicate how the RadioStation has changed since
 * last save or reset. Declared to use QFlags<> to ease flag
 * usage and to enforce type safety.
 */
enum ChangeFlag
{
    NoChange = 0,
    PersistentDataChanged = 1 << 0,
    NameChanged = 1 << 1,
    FavoriteChanged = 1 << 2,
    GenreChanged = 1 << 3,
    UrlChanged = 1 << 4,
    TypeChanged = 1 << 5,
    PCodeChanged = 1 << 6,
    PsTypeChanged = 1 << 7,
    RadioTextChanged = 1 << 8,
    DynamicPsChanged = 1 << 9
};
Q_DECLARE_FLAGS( Change, ChangeFlag )
/**
 * Flags to indicate station type.
 * Declared to use QFlags<> to ease flag usage and to
 * enforce type safety.
 */
enum TypeFlag
{
    Favorite = 1 << 0,
    LocalStation = 1 << 1,
    PreDefined = 1 << 2,
    ManualStation = 1 << 3
};
Q_DECLARE_FLAGS( Type, TypeFlag )
/**
 * Flag to indicate whether or not station uses dynamic PS
 * and if the check has been performed. Declared to use
 * QFlags<> to ease flag usage and to enforce type safety.
 */
enum PsTypeFlag
{
    Unknown,
    Dynamic,
    Static
};
Q_DECLARE_FLAGS( PsType, PsTypeFlag )
/**
 * Magical values used as preset indexes to signify certain
 * conditions. NotFound means that a find function could not
 * find a station. Invalid means that the station instance
 * has not been initialized. SharedNull identifies the empty
 * "null" station that every newly created station points to.
 */
enum PresetFlag { NotFound = -1,
    Invalid = -100,
    SharedNull = -200
};
/**
 * Static convenience function to parse a frequency.
 */
static QString parseFrequency( uint frequency );
RadioStation();
RadioStation( const RadioStation& other );
~RadioStation();
RadioStation& operator=( const RadioStation& other );
private:
  explicit RadioStation( int presetIndex, uint frequency );

  void reset();
  void setChangeFlags( Change flags );

  // Setters for persistent data.
  void setPresetIndex( int presetIndex );
  void setFrequency( uint frequency );
  void setName( const QString& name );
  void setGenre( const int genre );
  void setUrl( const QString& url );
  bool setPiCode( int piCode, RadioRegion::Region region );

  // Setters for non-persistent data.
  void setPsType( PsType psType );
  void setRadioText( const QString& radioText );
  void setDynamicPsText( const QString& dynamicPsText );

public: // Getters and setters.

  // Getters and setters for persistent data.
  bool isValid() const;
  QString name() const;
  void setuserDefinedName( const QString& name );
  bool isRenamed() const;
  int genre() const;
  QString frequencyString() const;
  uint frequency() const;
  int presetIndex() const;
  void setFavorite( bool favorite );
  bool isFavorite() const;
  QString url() const;
  int piCode() const;
  void setType( RadioStation::Type );
  void unsetType( RadioStation::Type );
  bool isType( RadioStation::Type ) const;

  // Convenience checkers.
  inline bool hasPiCode() const { return piCode() != -1; } 
  inline bool hasName() const { return !name().isEmpty(); } 
  inline bool hasUrl() const { return !url().isEmpty(); } 
  inline bool hasRadioText() const { return !radioText().isEmpty(); } 
  inline bool hasDynamicPs() const { return !dynamicPsText().isEmpty(); } 
  inline bool hasGenre() const { return genre() != -1; } 

  // Getters for non-persistent data.
  PsType psType() const;
  QString radioText() const;
  QString dynamicPsText() const;
  Change changeFlags() const;
  bool hasDataChanged( Change flags ) const;
  bool hasChanged() const;
  void resetChangeFlags();
  bool hasSentRds() const;
private:

// Methods for converting PI code into call sign.
QString piCodeToCallSign( uint programmeIdentification );
QString iterateCallSign( int piBase, int programmeIdentification );
char callSignChar( uint decimalValue );

private: // data

/**
 * Pointer to the implicitly shared private implementation.
 * Owned.
 */
QSharedDataPointer<RadioStationPrivate> mData;

public:

/**
 * Checks if the class is detached from implicitly shared data.
 * Required by many Qt convenience functions for implicitly shared classes.
 */
bool isDetached() const;

typedef QSharedDataPointer<RadioStationPrivate> DataPtr;
inline DataPtr& data_ptr() { return mData; }

Q_DECLARE_OPERATORS_FOR_FLAGS( RadioStation::Change )
Q_DECLARE_OPERATORS_FOR_FLAGS( RadioStation::Type )

// Can be moved around in memory by containers if necessary
Q_DECLARE_TYPEINFO( RadioStation, Q_MOVABLE_TYPE );

// Uses implicit sharing
Q_DECLARE_SHARED( RadioStation )

// To be usable in a QVariant
Q_DECLARE_METATYPE( RadioStation )

#endif // RADIOSTATION_H
Appendix 3. Images of the radio application running in Windows.

Figure 15 shows the radio application running in Windows on top of a stubbed wrapper layer. The left image is a rough mockup that contains the required features to help development but does not follow any UI design. The buttons to simulate various events are seen in the top console. The logo of the third party song recognition application has been covered with a black dot. The UI has been created with Orbit. The right image is one of the numerous QML UI proposals that got rejected. The UI has been created without Orbit.

Figure 15. Images of the radio running in Windows.
Appendix 4. The Kudos award received for the engine design work.

The device project name is not public information and has been covered with a black bar. Similarly, the names of people have been covered with red bars. Symbian Applications (SYMBA) was the name of the organization inside Nokia that we all belonged to. In the Nokia organization, the radio team was a part of the larger Videos SW Area even though we had nothing to do with video playback.

Figure 16. Kudos award for the engine design work.