|  |
| --- |
|  |
| Application Settings Design |
|  |
|  |
| **Andriy Kuznetsov** |

Document History

Revision History

| Author | Version | Rev. date | Summary of Changes | Section | Page |
| --- | --- | --- | --- | --- | --- |
| Andriy Kuznetsov | 1.0 | Apr 29, 2015 | Draft |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Table of Contents

[1 Introduction 3](#_Toc418110747)

[2 High level design (System Architecture) 3](#_Toc418110748)

[3 Low level designs (Sub-system architecture) 4](#_Toc418110749)

# Introduction

This document describes library which provides mechanisms to access application settings which are stored in INI-files.

Here are listed high-level requirements which were considered during development:

**REQ-1:** There shall exist settings that can be modified and saved by the user.

**REQ-2:** There shall exist user scope settings that can be modified and saved by the user.

**REQ-3:** User scope settings must only be available user dependent. No other user can read or modify other user settings.

**REQ-4:** there shall exist machine scope settings that can be modified and saved by the user

**REQ-5:** The machine scope settings must be made available for all users of the system. Any changes made by one user will be visible and applicable to other users.

**REQ-6:** The settings are available at the time that the software loads.

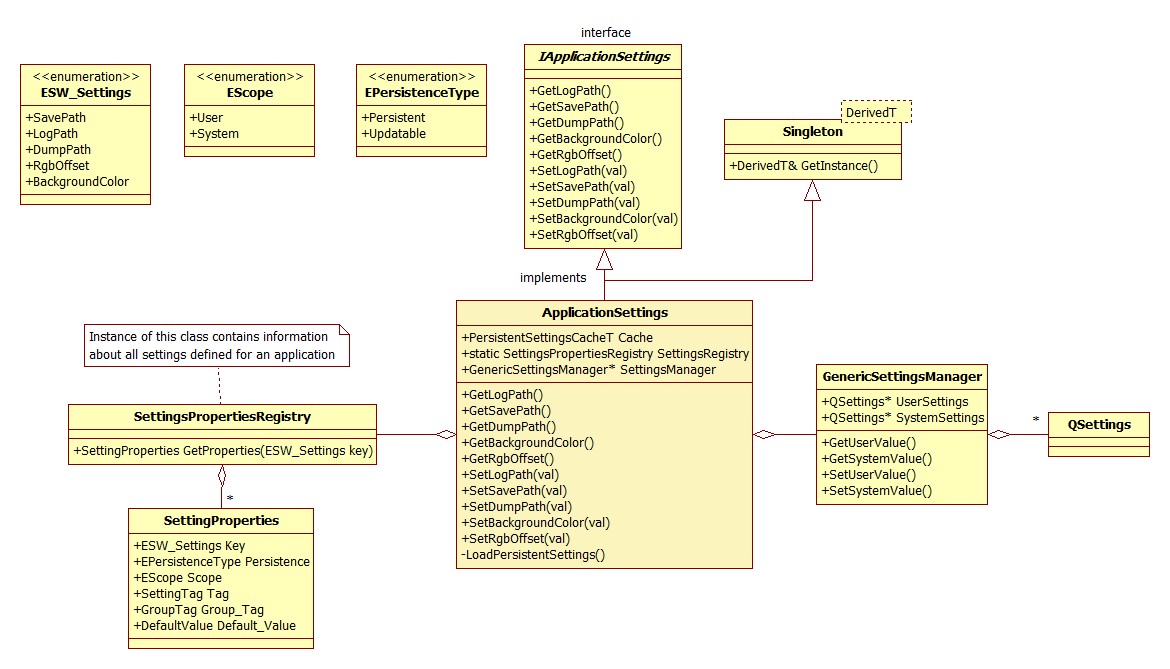
**REQ-7:** The settings are available at any given time of the software’s state or life.

# High level design (System Architecture)

The implementation is based on using QSettings class from Qt library which provides cross-platform mechanisms of accessing application settings considering different formats of settings storage – INI files, Registry, User-defined formats.

In addition, QSettings guarantees thread-safety of access operations, synchronization of settings values between different applications which use the same storage concurrently, and smart merging algorithms which handle concurrent updates of settings by different applications.

Here is the UML diagram which shows main classes involved in settings management.



The entry-point to settings management is **ApplicationSettings** class. It is responsible for addressing user read/write requests to application settings. **ApplicationSettings** represents the settings storage for particular application (in this case, My180Cam) – this means that it knows about all settings inherent for the application, and this property is reflected in **ApplicationSettings** interface.

# Low level designs (Sub-system architecture)

**ApplicationSettings**

**ApplicationSettings** is implemented as Singleton which allows developer to use its instance in any place of application codebase. But, to support loose coupling of application components, developer can use interface **IApplicationSettings** which defines interface implemented by **ApplicationSettings**.

Basic case of **ApplicationSettings** usage:

*typedef std::map<QString, QVariant> SettingsCacheT;*

*SettingsCacheT GetAppSettingsValues()*

*{*

*SettingsCacheT result;*

*ApplicationSettings& settings =* ***ApplicationSettings::GetInstance();***

*// Read settings and feed them to result…*

*return result;*

*}*

Another **ApplicationSettings** usage case which demonstrates how to avoid tight coupling between application components

/\* \*

Here we pass pointer to interface – thus, we do not depend on concrete implementation of

seetings management component

\*/

*SettingsCacheT GetAppSettingsValues( IApplicationSettings\* app\_settings )*

*{*

*SettingsCacheT result;*

*// Read settings and feed them to result*

*return result;*

*}*

**SettingsPropertiesRegistry**

Thisclass stores knowledge about exact set of settings inherent by My180Cam application.

Beside settings keys used to extract values from the storage (INI file), **SettingsPropertiesRegistry** stores properties of each setting.

There are several properties which describe each setting:

1. Persistence type.

Settings can by Updatable and Persistent. **Persistent** setting value is not impacted by modifications performed by other processes running concurrently and working with the same settings storage. **Updatable** settings value can by updated due to changes done in another application

1. Scope

Different settings can have different scopes. There are two scopes – **System** and **User**.

System settings are shared between all users of the PC, and User settings are duplicated for each user and one user cannot access other user’s settings.

1. Setting name and Group name

Setting and group names correspond to those in INI file.

1. Default value

Default setting value assigned to setting in case it was not successfully read from INI-file.

**SettingsPropertiesRegistry** can be extended to store more settings. In fact, it is not good solution to store this data hard-coded, it should be stored externally (in XML) and loaded during runtime so that adding or changing settings properties doesn’t demand application rebuild.

**GenericSettingsManager**

This class provides basic settings management features like reading and writing setting values. It doesn’t contain knowledge about exact INI file location or settings for the application. **GenericSettingsManager** implementation is based on using QSettings – it stores references to two QSettings instances, one is responsible to access User settings, another – System settings.

**ApplicationSettings** class uses **GenericSettingsManager** as settings access provider. It configures settings manager to work with exact settings storage.