СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 1](#_Toc197185546)

[ВВЕДЕНИЕ 2](#_Toc197185547)

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 3](#_Toc197185548)

[1.1. Обоснование актуальности работы 3](#_Toc197185549)

[1.2. Обзор существующих решений 3](#_Toc197185550)

[1.3. Анализ существующих решений 4](#_Toc197185551)

[QSettings (Qt Core) 5](#_Toc197185552)

[Qt.labs.settings (QML) 5](#_Toc197185553)

[KConfig (KDE Framework) 5](#_Toc197185554)

[Заключение 6](#_Toc197185555)

[ГЛАВА 2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ 8](#_Toc197185556)

[2.1. Функциональные требования 8](#_Toc197185557)

[2.2. Нефункциональные требования 8](#_Toc197185558)

[ГЛАВА 3. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ 8](#_Toc197185559)

[ЗАКЛЮЧЕНИЕ 8](#_Toc197185560)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 8](#_Toc197185561)

ВВЕДЕНИЕ

Почти в каждом десктопном приложении в том или ином виде присутствует окно настроек. При разработке таких приложений на C++ и Qml создание таких настроек с нуля является очень трудоёмкой задачей. Если не создать отдельный модуль для этой задачи – то может появиться огромное количество boilerplate кода, который очень сильно затруднит поддержку и развитие в любой области разработки. Поэтому нашей компанией было принято решение создать такую библиотеку, сделать её максимально обобщенной и гибкой в использовании.

ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

Перед тем как на самом деле приступать к задаче, необходимо обратиться к открытым источникам и выяснить, не написал ли кто-нибудь уже такой же или подобный проект чтобы использовать его или хотя бы взять за основу для создания нового.

1.1. Обоснование актуальности работы

Как уже было упомянуто во введении – подобная работа может очень сильно упростить и ускорить разработку приложений и избавить от лишнего написания кода. Описывать QML-интерфейс для каждого параметра и каждый раз писать логику загрузки/сохранения настроек может очень сильно усложнить процесс разработки. Благодаря библиотеке достаточно будет только предоставить json с описанием настроек, не будет необходимости в перекомпиляции и написании своих компонентов. При обнаружении ошибок в коде библиотеки достаточно будет лишь в одном месте их поправить, и исправления применятся во всех проектах которые используют данный модуль.

1.2. Обзор существующих решений

Для начала рассмотрим класс **QSettings**. Класс является частью большого популярного фреймворка Qt. QSettings позволяет сохранять и загружать настройки из ini файлов, реестра Windows и других платформенно зависимых хранилищ. Тем самым мы имеем кроссплатформенный способ манипулирования настроек. Однако есть очень важные замечания, связанные с этим решением. Во-первых, это всего лишь backend часть для настроек. Во-вторых, мы ограничиваемся возможностями, реализованными в QSettings. То есть мы не сможем сериализовать свои типы данных, поскольку QVariant содержит предопределенные типы.

Далее рассмотрим **Qt.labs.settings (QML)**, Qml модуль который предоставляет доступ к QSettings внутри qml. С этим модулем проблема ровно та же самая, что и с просто классом QSettings, то есть надо писать интерфейс самому.

Также существует фреймворк **KConfig (KDE Framework)**. Наверное, из названия сразу же становится понятно почему это может стать очень плохим решением для проекта. Используя KConfig мы создаём прямую зависимость от KDE, что означает полную скованность и невозможность запустить проект например на операционной системе **Windows**, на которой поддержки KDE, конечно же, не будет.

Ну и в завершение обзора существующих решений упомянем альтернативный вариант, когда мы просто не создаем библиотеку и каждый раз в проекте пишем своё решение для настроек. В таком случае мы получаем полное отсутствие стандартизации и, как говорится, каждый раз «изобретаем велосипед». Такой подход к программированию очень быстро погубит архитектуру существующей системы и безусловно приведет к неподдерживаемым проектам.

1.3. Анализ существующих решений

Рассмотрев все имеющиеся решения, их недостатки и достоинства, можно сделать вывод, что предложенная библиотека действительно имеет место быть. Но для эффективного анализа необходимо ввести критерии чтобы полноценно провести анализ.

Критерии оценивания:

1. Гибкость – возможность работы с разными типами данных (строки, числа, списки, вложенные структуры).
2. Интеграция с QML – насколько просто подключить решение в QML-проект без переписывания кода.
3. Автоматизация UI – требуется ли ручное создание интерфейса или он генерируется автоматически.
4. Формат хранения – поддержка JSON, INI, бинарных форматов и т. д.
5. Расширяемость – возможность добавления новых типов настроек и кастомизации.
6. Зависимости – наличие сторонних библиотек, усложняющих развёртывание.

QSettings (Qt Core)

**Плюсы:**

* Встроен в Qt, не требует дополнительных зависимостей.
* Поддерживает кроссплатформенное хранение (INI, реестр Windows, plist macOS).
* Простой API для сохранения/загрузки примитивных типов.

**Минусы:**

* Нет встроенного механизма генерации UI – интерфейс нужно писать вручную.
* Ограниченная поддержка сложных структур (например, вложенных объектов).
* Нет валидации и описаний параметров.

Таким образом, данное решение подходит для простых случаев, но не решает задачу автоматизации интерфейса настроек.

Qt.labs.settings (QML)

**Плюсы:**

* Позволяет работать с настройками напрямую из QML.
* Интегрируется с QSettings, сохраняя кроссплатформенность.

**Минусы:**

* Только базовые типы данных (нет массивов, сложных объектов).
* Интерфейс всё равно создаётся вручную.
* Не поддерживает динамическое обновление схемы настроек.

Так, данное решение удобно для минималистичных проектов, но не подходит для сложных конфигураций.

KConfig (KDE Framework)

**Плюсы:**

* Поддержка сложных структур, групп настроек и зависимостей.
* Возможность удалённого управления конфигурацией.

**Минусы:**

* Зависит от KDE Frameworks, что неприемлемо для многих Qt-проектов.
* Ориентирован на C++/QtWidgets, интеграция с QML затруднена.
* Избыточность для большинства приложений.

Мощный инструмент, но слишком тяжёлый и специализированный для обычных задач.

Заключение

Анализ показал, что существующие решения либо слишком примитивны (QSettings, Qt.labs.settings), либо требуют значительных доработок (самописные системы), либо перегружены ненужной функциональностью (KConfig).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Решение | Гибкость | QML интеграция | Авто-UI | Формат хранения | Расширяемость | Зависимости |
| QSettings | Низкая | Частичная | Нет | INI/реестр | Нет | Qt Core |
| Qt.labs.settings | Низкая | Да | Нет | INI/реестр | Нет | Qt Quick |
| Самописные | Высокая | Да | Частично | JSON/другие | Да | Нет |
| KConfig | Очень высокая | Нет | Нет | Свой | Да | KDE |
| Новое решение | Высокая | Да | Да | JSON/другие | Да | Только Qt |

Таблица 1 - Сравнение доступных решения

Преимущества предлагаемой библиотеки:

1. Автоматизация – генерация UI из JSON без ручного описания QML.
2. Гибкость – поддержка вложенных структур, динамических списков и кастомных типов.
3. Простота интеграции – минимальный код для подключения в QML-проект.
4. Отсутствие лишних зависимостей – только стандартные компоненты Qt.

Таким образом, разработка данной библиотеки оправдана отсутствием готовых решений, сочетающих удобство, гибкость и лёгкость интеграции в QML-приложения.

ГЛАВА 2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ

2.1. Функциональные требования

2.2. Нефункциональные требования

ГЛАВА 3. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ