Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности

Высшая школа программной инженерии

Работа допущена к защите

Директор ВШПИ

\_\_\_\_\_\_\_\_\_\_\_\_\_ П.Д.Дробинцев

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_2025 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**работа бакалавра**

**РАЗРАБОТКА БИБЛИОТЕКИ НА С++/QML ДЛЯ ДИНАМИЧЕСКИ ГЕНЕРИРУЕМОГО ОКНА НАСТРОЕК**

по направлению подготовки (специальности)

**09.03.04 Программная инженерия**

Направленность (профиль)

**09.03.04\_01 Технология разработки и сопровождения качественного программного продукта**

Выполнил студент гр. А.М. Абраамян

5130904/10101

Руководитель Т.В. Коликова

Страший преподаватель ВШПИ

Консультант

по нормконтролю

Санкт-Петербург

2025

Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности

Высшая школа программной инженерии

УТВЕРЖДАЮ

Директор ВШПИ

П. Д. Дробинцев

«\_\_» \_\_\_\_\_\_\_\_\_\_ 2025 г.

**ЗАДАНИЕ**

**на выполнение выпускной квалификационной работы**

студенту Абраамяну Александру Манвеловичу, группа 5130904/10101

1. Тема работы: Разработка библиотеки на С++/QML для динамически генерируемого окна настроек

2. Срок сдачи студентом законченной работы: 17.05.2025

3. Исходные данные по работе: В результате анализа существующих решений для управления настройками приложений выявлена необходимость разработать гибкую библиотеку на QML (Qt Quick), интегрируемую через Conan, которая автоматически генерирует UI окна настроек на основе входного JSON-файла. Такое решение позволит избежать дублирования кода во многих проектах в части связанной с настройками приложения. Основные задачи: динамическое создание UI, легкость и гибкость конфигурации, сохранение и загрузка, кастомизация UI

4. Содержание работы (перечень подлежащих разработке вопросов):

* Обоснование значимости темы и обзор исследуемой области
* Исследование существующих решений для генерации UI из JSON-файла
* Предлагаемый подход к разработке библиотеки
* Выбор и обоснование технологического стека
* Разработка и тестирование прототипа
* Подготовка документации по использованию проекта
* Выводы и рекомендации по внедрению и развитию

5. Перечень графического материала (с указанием обязательных чертежей):

6. Перечень используемых информационных технологий, в том числе программное обеспечение, облачные сервисы, базы данных и прочие сквозные цифровые технологии (при наличии):

* GCC
* Clang
* MSVC
* QML (UI-описание)
* CMake (скрипты сборки)
* Qt 5 (модули: Core, Gui, QML, Quick, Quick Controls 2)
* Conan (менеджер зависимостей для C++/Qt)
* CMake (основная система сборки)
* Catch2 (модульные тесты C++)
* GitLab CI (CI/CD)
* Doxygen (C++ документация)
* Conan Center / Artifactory (хостинг пакетов Conan)

7. Консультанты по работе:

8. Дата выдачи задания: 14.04.2025



Руководитель ВКР Коликова Т. В.

(подпись)

Задание принял к исполнению 14.04.2025



Студент Абраамян А. М.

(подпись)

РЕФЕРАТ

60 страниц, 11 рисунков, 1 таблица

КЛЮЧЕВЫЕ СЛОВА: Qt/QML, GNU, Clang, msvc, JSON, автогенерация UI, Conan, Artifactory, C++, CMake, Catch2, fmt, jsoncpp, spdlog

Тема выпускной квалификационной работы: «Разработка библиотеки на С++/QML для динамически генерируемого окна настроек»

Целью этой работы было создание библиотеки для автоматической генерации окна настроек в соответствии с предоставленным описанием в виде JSON для ускорения и упрощения разработки настольных приложений. В рамках работы были решены следующие задачи:

1. Анализ известных и доступных решений в выбранной сфере
2. Анализ доступных способов поставки итогового продукта
3. Разработка гибкой и надежной архитектуры backend части
4. Проектирование frontend части приложения

Для реализации проекта были выбраны следующие технологии: язык программирования C++ (стандарт 14), среда разработки Visual Studio Code, система управления зависимостями conan.

В итоге было успешно реализовано решение, обеспечивающее упрощенную разработку настольных приложений и выполняющее все поставленные в ТЗ требования.

ABSTRACT

60 pages, 11 pictures, 1 table

KEYWORDS: Qt/QML, GNU, Clang, msvc, JSON, autogeneration of UI, Conan, Artifactory, C++, CMake, Catch2, fmt, jsoncpp, spdlog

The topic of the graduation project: «Developing a C++/QML library for a dynamically generated settings window»

The purpose of this work was to create a library for automatically generating a settings window in accordance with the provided description in the form of JSON to speed up and simplify the development of desktop applications. As part of the work, the following tasks were solved:

1. Analysis of known and available solutions in the chosen field
2. Analysis of available delivery methods for the final product
3. Development of a flexible and reliable backend architecture
4. Designing the frontend part of the application

The following technologies were selected for the project: the C++ programming language (standard 14), the Visual Studio Code development environment, and the conan dependency management system.

As a result, a solution was successfully implemented that provides simplified desktop application development and fulfills all the requirements set out in the TOR.

СОДЕРЖАНИЕ

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 7](#_Toc198473276)

[1.1. Обоснование актуальности работы 7](#_Toc198473277)

[1.2. Обзор существующих решений 7](#_Toc198473278)

[1.3. Анализ существующих решений 8](#_Toc198473279)

[1.3.1 QSettings (Qt Core) 9](#_Toc198473280)

[1.3.2 Qt.labs.settings (QML) 9](#_Toc198473281)

[1.3.3 KConfig (KDE Framework) 9](#_Toc198473282)

[Заключение 10](#_Toc198473283)

[ГЛАВА 2. ВЫБОР СПОСОБА ПОСТАВКИ БИБЛИОТЕКИ 10](#_Toc198473284)

[2.1. Установка системной зависимости 11](#_Toc198473285)

[2.2. Использовать git submodule 14](#_Toc198473286)

[2.3. Использовать пакетный менеджер conan 17](#_Toc198473287)

[ГЛАВА 3. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ 20](#_Toc198473288)

[3.1. Функциональные требования 20](#_Toc198473289)

[3.2. Нефункциональные требования 22](#_Toc198473290)

[ГЛАВА 4. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ 24](#_Toc198473291)

[4.1. Структура JSON 24](#_Toc198473292)

[4.1.1. Boolean тип 24](#_Toc198473293)

[4.1.2. Integer/Real/UInt типы 24](#_Toc198473294)

[4.1.3. String тип 25](#_Toc198473295)

[4.1.4. Array тип 25](#_Toc198473296)

[4.1.5. FilePath тип 25](#_Toc198473297)

[4.1.6. Selection тип 25](#_Toc198473298)

[4.1.7. Coordinate тип 25](#_Toc198473299)

[4.2. Основные классы в C++ 26](#_Toc198473300)

[4.2.1. Класс ISetting 26](#_Toc198473301)

[4.2.2. Класс ISelectionSetting 27](#_Toc198473302)

[4.2.3. Класс IRegexStringSetting 27](#_Toc198473303)

[4.2.4. Прочие классы 27](#_Toc198473304)

[4.2.5. Фабричные классы ISettingParser ISettingsGroupParser ISettingsJsonParser 27](#_Toc198473305)

[4.2.6. Model класс TabsModel 29](#_Toc198473306)

[4.2.7. Model класс SettingsListModel 30](#_Toc198473307)

[4.2.8. Класс ISettingValue 31](#_Toc198473308)

[4.2.9. Класс SimpleGetSetValueMixin 31](#_Toc198473309)

[4.3. Основные QML элементы 32](#_Toc198473310)

[4.3.1 SettingsEditor.qml 32](#_Toc198473311)

[4.3.1 SettingsTable.qml 35](#_Toc198473312)

[4.3.2. DelegateManager.qml 36](#_Toc198473313)

[4.3.3. SettingsEditorDialog.qml и SettingsEditorWindow.qml 36](#_Toc198473314)

[Заключение 37](#_Toc198473315)

[ГЛАВА 5. АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ 37](#_Toc198473316)

[ГЛАВА 6. ПЕРСПЕКТИВЫ РАЗВИТИЯ 38](#_Toc198473317)

[ЗАКЛЮЧЕНИЕ 40](#_Toc198473318)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 41](#_Toc198473319)

[ПРИЛОЖЕНИЕ 42](#_Toc198473320)

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[10]. 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)[10], 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)[11]. 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.

1.3.1 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.

1.3.2 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.

1.3.3 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. ВЫБОР СПОСОБА ПОСТАВКИ БИБЛИОТЕКИ

Разработчикам доступно много способов поставки библиотеки. Однако очень важно сразу выбрать правильный, чтобы было удобно пользоваться итоговым продуктом и не было никаких неприятностей. Основные проблемы, которые призван решать этот раздел – каким способом поставить конечный продукт и каким способом поставить зависимости конечного продукта. Предварительно можно назвать как минимум три зависимости конечного продукта.

* Jsoncpp – библиотека для эффективной работы с json файлами
* Fmtlib[9] – библиотека для удобного форматирования строк
* Spdlog[8] – библиотека для быстрого логирования

Три самых популярных способа поставки библиотек в области С++:

* Установить зависимость системно
* Подтянуть в качестве git submodule проекта
* Использовать пакетный менеджер conan

Необходимо проанализировать каждый, узнать какие у него плюсы, минусы и критические проблемы.

2.1. Установка системной зависимости

Начать стоит с системных зависимостей. На Debian-based дистрибутивах можно легко поставить, например, fmtlib с помощью пакетного менеджера apt командой apt install libfmt-dev и дальше спокойно им пользоваться. Но одним из требований к проекту является кросс платформенность.

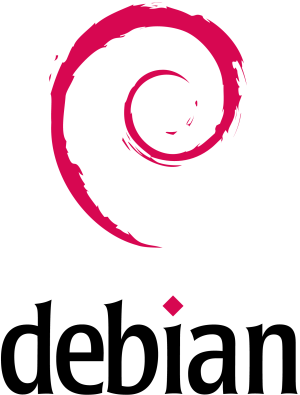


Рисунок 1 - логотип Debian

Итак, рассмотрим случай. Допустим мы разрабатываем библиотеку на Ubuntu 22.04 LTS. Поставили libfmt-dev пакет, он оказался версии 10.1.1, мы пишем #include <fmt/format.h> и спокойно разрабатываем нашу библиотеку.

Далее мы взяли свежий Debian 12 образ, хотим на нём использовать нашу библиотеку, ставим пакет libfmt-dev 10.1.1, всё работает, всё отлично. Мы можем спокойно пользоваться нашим проектом.

Потом коллега на дистрибутиве линукса Manjaro внезапно тоже захотел поработать с нашим проектом, читает README и видит, что необходимо поставить библиотеку libfmt. Пишет команду sudo pacman -S fmt и получает версию библиотеки 11.1.4. Оказалось, что в этой версии библиотеки заголовочный файл fmt/format.h заменили на fmt/core.h, который был в версии libfmt 10.1.1.

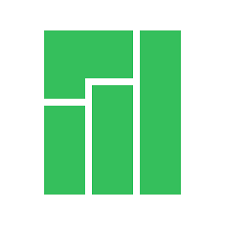


Рисунок 2 - логотип Manjaro

Чтобы обойти эту проблему можно использовать, например, следующий код.

#if FMT\_VERSION >= 110000 // 11.0.0+

#include <fmt/core.h>

#else

#include <fmt/format.h>

#endif

Не нужно быть большим экспертом чтобы понять, что такой подход приведёт в никуда. Неужели так придется делать с каждой библиотекой? А что если API библиотеки поменяется? А как тестировать такой проект? Можно было бы полностью инкапсулировать версию fmt в отдельный модуль, но тогда программисты вместо того чтобы писать код для проекта – большую часть времени занимались бы решением проблем совместимости.

Но это еще не конец истории. Далее пришло задание собрать проект, использующий нашу библиотеку, на платформе Windows. Там и вовсе никакого пакетного менеджера нету, так что придется собирать и ставить libfmt полностью вручную, что очень сильно усложняет поставку библиотеки.

Таким образом, вместо «Нам нужно поддерживать две разные версии fmt» необходимо задать вопрос: «Почему мы вообще допускаем такое неконтролируемое разнообразие версий?».

Теперь проблема с системными зависимостями видна и становится понятно, что нельзя использовать этот способ для поставки библиотек.

2.2. Использовать git submodule



Рисунок 3 - логотип Git Submodule

Альтернативой системным библиотекам является git submodule. Этот инструмент позволяет вкладывать одни репозитории в другие. Таким образом разработчики смогут полностью поместить весь исходный код libfmt и зафиксировать нужную версию прямо у себя в проекте. Делается это с помощью следующий команд:

Добавить библиотеку (spdlog, googletest, fmt)

* git submodule add

Обновить до свежей версии

* git pull
* git submodule update —init —recursive

Выбрать версию

* git checkout v0.1.0

Такой подход имеет явное преимущество перед системными библиотеками. Но это всё еще не идеально. Система Git Submodules содержит несколько очень разрушительных проблем, которые всё еще не позволят благополучно использовать нашу библиотеку.

Представим ситуацию: в проекте есть две библиотеки, LibA и LibB, и обе зависят от одной и той же LibX, но требуют разных её версий. Если подключить их через Git Submodules, то CMake выберет только одну версию LibX — ту, которая встретилась первой. Вторая версия просто проигнорируется, и при этом не будет никакой ошибки компиляции.

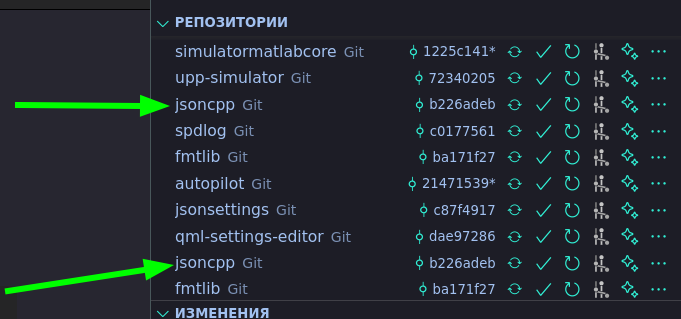


Рисунок 4 - проблема разных версий библиотек

Проблема в том, что система сборки не видит конфликта. Она находит LibX в одном из подмодулей и использует её, даже если другая часть проекта ожидает совершенно другую версию. В результате возможны трудноотлавливаемые ошибки:

* Несовместимые изменения API в разных версиях LibX могут приводить к падениям в рантайме.
* Ошибки линковки, если символы из одной версии LibX конфликтуют с другой.
* Неочевидные баги, когда код работает в одних условиях и ломается в других.

Если зависимости сами используют подмодули (а их зависимости — свои подмодули), структура проекта быстро превращается в "матрешку". Это усложняет:

* Клонирование проекта — нужно рекурсивно подтягивать все подмодули, и если какой-то из них недоступен (например, приватный репозиторий), процесс ломается.
* Обновление зависимостей — если LibX обновилась в одном месте, её нужно вручную синхронизировать во всех подмодулях, где она используется.
* Размер репозитория — история подмодулей может занимать много места, даже если сам проект небольшой.

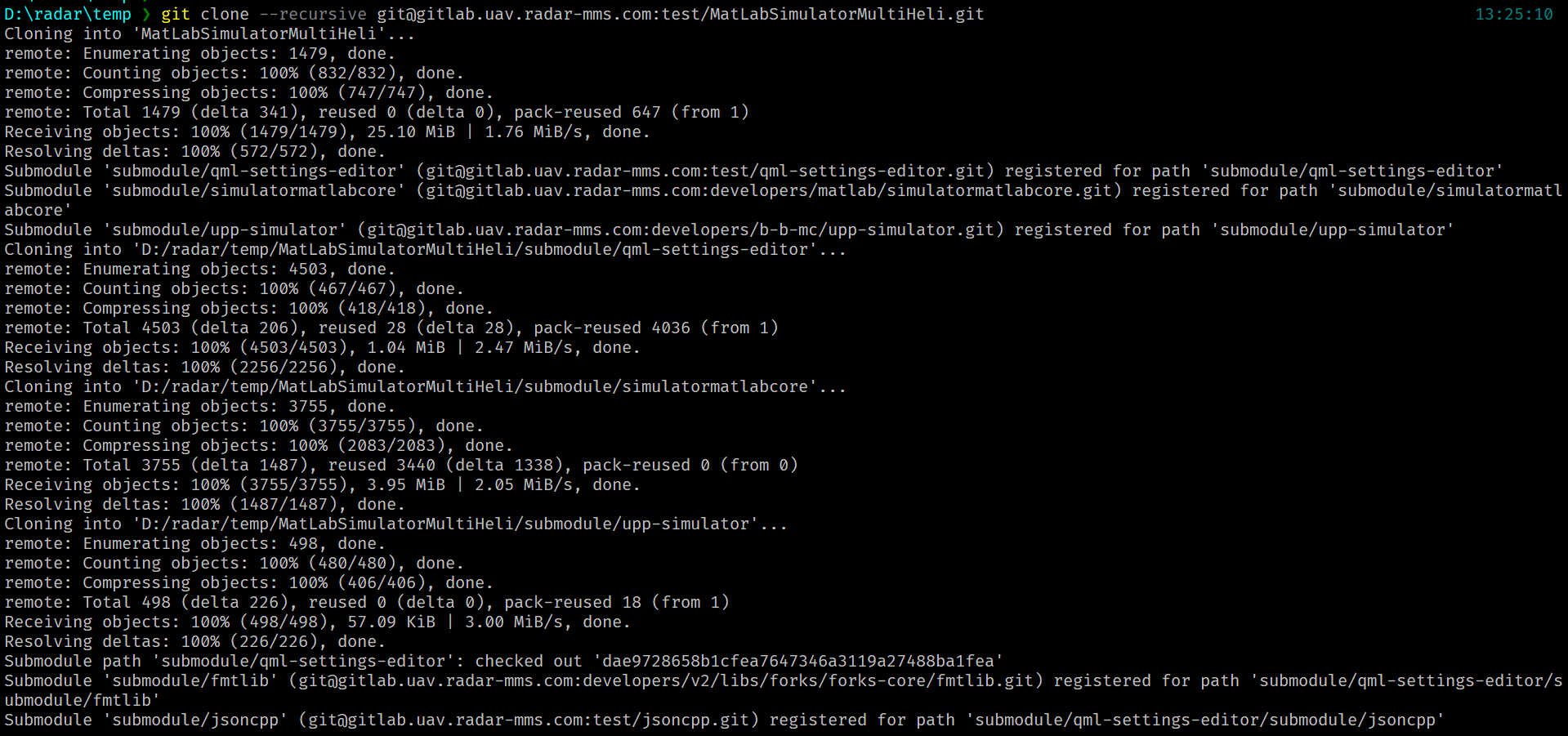


Рисунок 5 - пример клонирования проекта с сабмодулями

В отличие от пакетных менеджеров, где зависимости чётко указываются в конфигурационном файле, в подмодулях версии зафиксированы лишь хешами коммитов. Это значит:

* Невозможно быстро проверить, какие версии библиотек используются — нужно вручную заглядывать в каждый подмодуль.
* Сложно автоматизировать обновления — нет механизма "обнови все зависимости до последних стабильных версий".
* Риск "поломать" проект, если обновление подмодуля задевает другие части системы.

При использовании submodules каждая зависимость включается в проект в виде исходного кода и компилируется непосредственно в его рабочей директории. Это создаёт несколько проблем:

Если несколько подмодулей зависят от одной библиотеки (например, zlib или fmt), она может компилироваться несколько раз — по одному для каждого подмодуля. Это не только увеличивает время сборки, но и расходует дисковое пространство.

В отличие от систем управления пакетами, где скомпилированные библиотеки сохраняются в кэше, submodules требуют полной пересборки всех зависимостей при каждом обновлении или изменении конфигурации.

Поскольку системы непрерывной интеграции обычно начинают сборку с чистого состояния, они вынуждены каждый раз заново загружать и компилировать все подмодули, даже если изменения затронули лишь небольшую часть проекта.

Таким образом, использование Git Submodules для управления зависимостями существенно усложняет процесс сборки, увеличивает время разработки и создаёт дополнительные риски при обслуживании проекта. В отличие от современных пакетных менеджеров, submodules не обеспечивают кэширование, контроль версий и надёжное восстановление зависимостей, что делает их менее предпочтительным выбором для серьёзных проектов.

2.3. Использовать пакетный менеджер conan

В контексте данной работы использование системы управления пакетами Conan[3] было выбрано в качестве основного инструмента.



Рисунок 6 - логотип пакетного менеджера conan

Во-первых, Conan предоставляет механизм точного контроля версий зависимостей. В отличие от Git Submodules, где версии библиотек фиксируются лишь хешем коммита и могут быть непрозрачными для разработчика, Conan позволяет явно указывать требуемые версии пакетов в файле conanfile.py. Это позволяет легко следить за версиями зависимостей и избегать проблем, связанных с использованием различной версии одной и той же библиотеки, тем самым ускоряя разработку и делая ее гораздо проще.

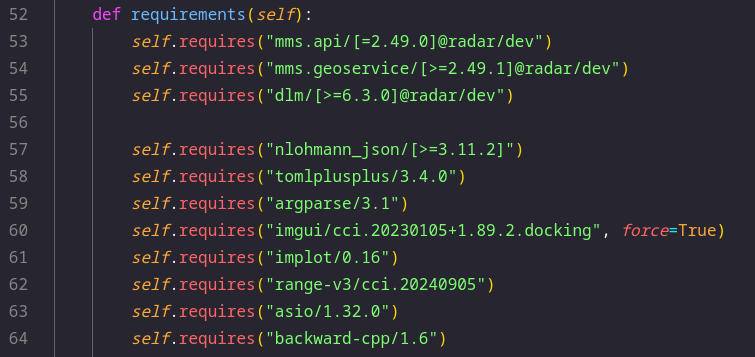


Рисунок 7 - пример conanfile.py

Во-вторых, Conan реализует эффективную систему кэширования собранных библиотек. При работе с подмодулями каждая зависимость компилируется непосредственно в дереве проекта, что при удалении директории сборки или изменении конфигурации приводит к необходимости повторной полной сборки всех зависимостей. Conan сохраняет скомпилированные пакеты в ~/.conan2/projects, что значительно ускоряет повторные сборки и делает процесс разработки более предсказуемым.

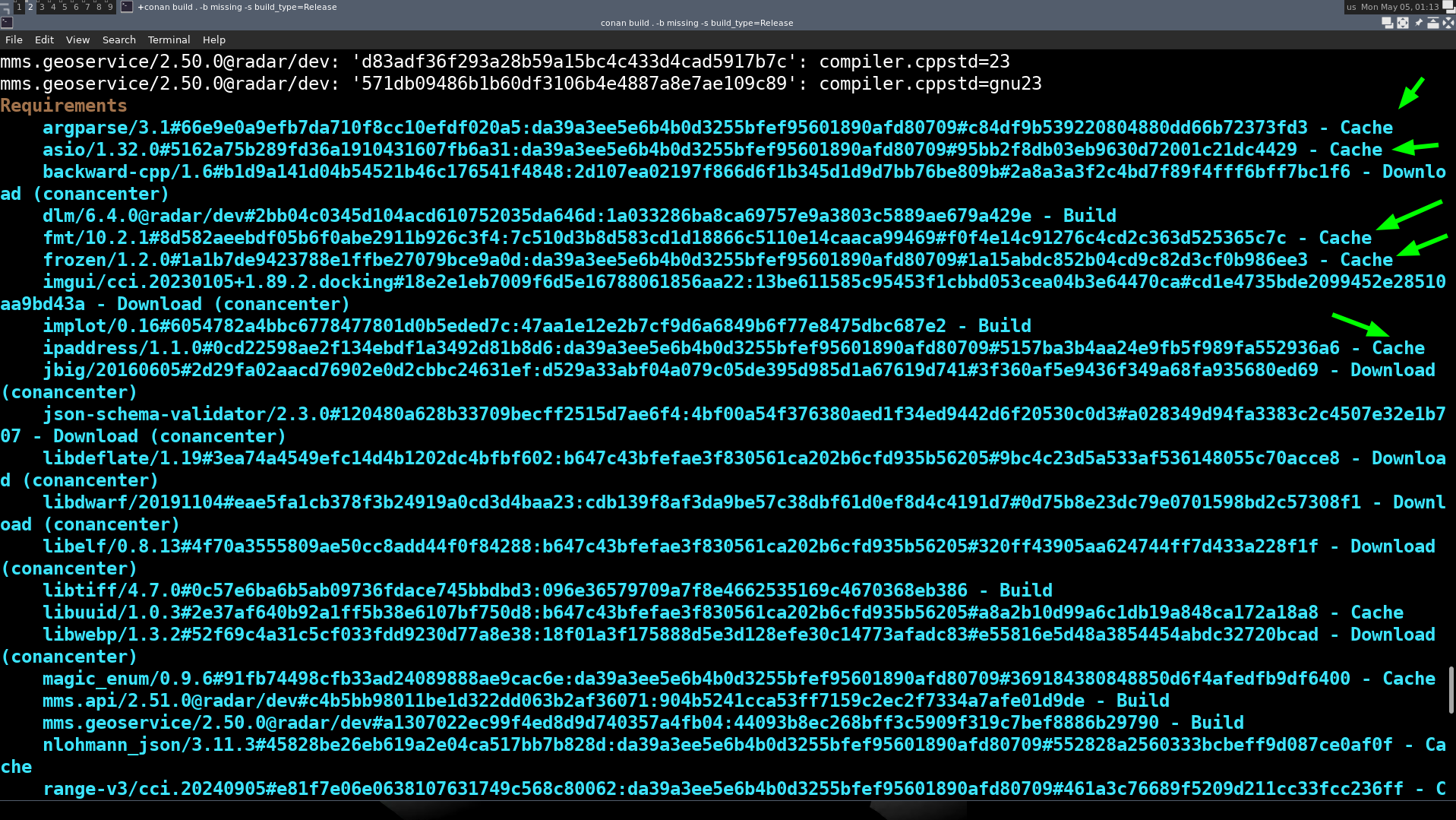


Рисунок 8 - прпимер кеширования библиотек

Важным преимуществом является также поддержка кроссплатформенности. Conan обеспечивает корректную работу зависимостей на разных операционных системах и с различными инструментами сборки, автоматически разрешая специфичные для платформы особенности. Это особенно ценно при разработке библиотеки, которая должна работать в разнородных окружениях.

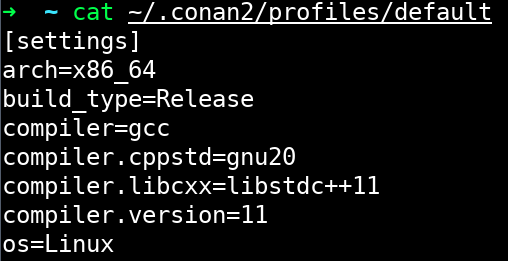


Рисунок 9 - пример профиля conan для обычной сборки проекта

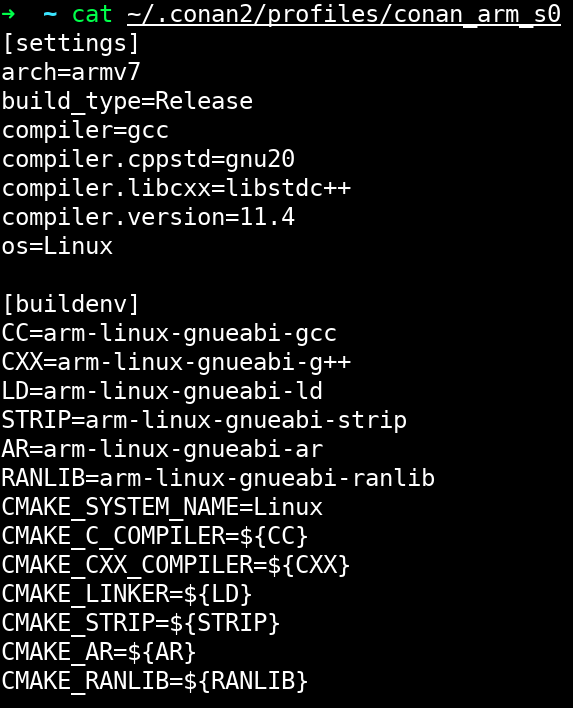


Рисунок 10 - пример профиля conan для кросс компиляции всех библиотек под arm

Кроме того, Conan предоставляет удобные механизмы для создания и распространения собственных пакетов. Это позволяет не только использовать внешние зависимости, но и легко упаковывать компоненты проекта для их повторного использования в других решениях. В контексте данной работы это означает возможность простого распространения разрабатываемой библиотеки среди потенциальных пользователей.

Наконец, интеграция Conan с современными системами сборки, такими как CMake, выполняется практически прозрачно для разработчика, сохраняя привычный рабочий процесс. При этом обеспечиваются все перечисленные преимущества в сравнении с альтернативными подходами к управлению зависимостями.

Таким образом, выбор Conan в качестве программы для управления зависимостями в проекте является хорошим выбором если нас интересует надежность, воспроизводимость сборок и удобство сопровождения кода. Этот подход соответствует современным практикам разработки программного обеспечения и позволяет избежать многочисленных проблем, характерных для более примитивных методов работы с внешними библиотеками.

ГЛАВА 3. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ

Хорошо составленные требования играют ключевую роль при разработке программного обеспечения. Для данной библиотеки это особенно важно поскольку проект является сложным, и чтобы не пришлось переписывать, необходимо учесть все важные детали разработки и реализации. Традиционно требования разделяют на функциональные и нефункциональные.

3.1. Функциональные требования

* Загрузка конфигурации
* Библиотека должна предоставлять возможность загрузки описания настроек из JSON-файла
* Поддерживаемый формат JSON-файла должен включать:
* Идентификатор настройки
* Тип настройки (строка, число, булево значение, список)
* Значение по умолчанию
* Отображаемое имя (для интерфейса)
* Дополнительные параметры (минимальное/максимальное значение для чисел, список вариантов для выбора)
* Генерация пользовательского интерфейса
* Автоматическое создание QML-интерфейса на основе загруженной конфигурации
* Поддержка следующих типов элементов управления:
* TextField для строковых значений
* SpinBox/Slider для числовых значений
* CheckBox для булевых значений
* ComboBox для выбора из списка
* Группировка настроек по секциям
* Управление значениями
* Возможность программного получения текущих значений настроек через API
* Механизм программного изменения значений настроек
* Валидация вводимых значений согласно ограничениям, указанным в конфигурации
* Сохранение и восстановление
* Функция сохранения текущих значений настроек в JSON-файл
* Возможность загрузки ранее сохраненных значений
* Поддержка сброса настроек к значениям по умолчанию
* Интеграционные возможности
* Предоставление сигналов QML об изменении значений настроек
* API для ручного управления настройками из C++ кода
* Возможность расширения списка поддерживаемых типов настроек
* Визуальная кастомизация
* Возможность изменения стилей элементов интерфейса через QML-свойства
* Поддержка тем оформления (светлая/темная)

3.2. Нефункциональные требования

* Требования к производительности
* Время загрузки конфигурации из JSON-файла размером до 100 КБ не должно превышать 50 мс на среднем оборудовании (CPU Intel Core i5 последнего поколения)
* Генерация интерфейса для 50+ настроек должна выполняться менее чем за 200 мс
* Операции сохранения/загрузки настроек должны выполняться асинхронно без блокировки UI-потока
* Требования к надежности
* Обработка ошибок при:
* Повреждении или неверном формате JSON-файла
* Отсутствии прав доступа к файлам конфигурации
* Попытке установки недопустимых значений
* Автоматическое восстановление при сбоях с сохранением работоспособности базового функционала
* Гарантированная целостность данных при сохранении настроек
* Требования к совместимости
* Поддержка Qt 5.15 и новее
* Кроссплатформенная работа на:
* Windows 10/11
* Linux (Ubuntu 20.04+, Fedora 35+)
* Совместимость с архитектурами x86\_64 и ARM64
* Требования к безопасности
* Валидация входных данных JSON-файла перед обработкой
* Ограничение максимального размера загружаемого JSON-файла (до 2 МБ)
* Шифрование чувствительных данных при сохранении (опционально)
* Требования к удобству использования
* Четкая документация API с примерами использования
* Подробные сообщения об ошибках для разработчиков
* Логирование ключевых событий (уровень DEBUG)
* Требования к сопровождению
* Модульная архитектура с четким разделением компонентов
* Полное покрытие unit-тестами критического функционала (минимум 80%)
* Поддержка CI/CD-процессов (автоматизированные сборки и тесты)
* Ограничения
* Максимальный размер JSON-конфигурации: 2 МБ
* Максимальное количество поддерживаемых настроек: 500
* Минимальные требования к оборудованию:
* 2 ГБ оперативной памяти
* 100 МБ свободного дискового пространства
* OpenGL 3.0+ для QML-интерфейса
* Требования к локализации
* Поддержка Unicode (UTF-8) для всех текстовых элементов
* Возможность добавления переводов без перекомпиляции
* Требования к ресурсоемкости
* Максимальное потребление памяти: 50 МБ при 100 активных настройках
* Средняя загрузка CPU в режиме ожидания: < 1%

ГЛАВА 4. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ

4.1. Структура JSON

Корневой элемент был спроектирован следующим образом:

{

"groups": [...]

}

Каждая группа состоит из следующих полей:

{

"name": "block\_1",

"caption": "Test Block Name 1",

"settings": [...]

}

* name – технический идентификатор
* caption – отображаемое название группы
* settings – массив параметров

Структура каждой настройки выбрана следующим образом:

{

"name": "unique\_id",

"caption": "Отображаемое имя",

"description": "Подсказка",

"type": "Тип",

"default\_value": ...,

"is\_required": false

}

Далее будут перечислены допустимые типа настроек, их особенности и соответствующие им элементы в UI.

4.1.1. Boolean тип

Этот тип может принимать значения true, false. UI элемента будет простой Checkbox.

4.1.2. Integer/Real/UInt типы

Эти типы представляют собой целочисленные и дробные значения. UI элемент будет простой TextField с регулярным выражением для валидации ввода пользователя.

4.1.3. String тип

Этот тип представляет собой простое поле для ввода текста. UI компонент будет также простым TextField.

4.1.4. Array тип

Данный тип должен поддерживать различные значения и типы. UI компонент будет в первую очередь ListView который будет внутри себя содержать делегаты настройки в соответствии с выбранным типом.

4.1.5. FilePath тип

Этот тип необходим для выбора файла из файловой системы компьютера. UI элементы FileDialog и TextField для выбора и отображения выбранного файла.

Json описания этой настройки имеет дополнительное поле extensions

4.1.6. Selection тип

Данный тип будет представлять собой выбор предопределенных элементов. UI элемент будет Combobox.

Json описания этой настройки имеет дополнительное поле options

4.1.7. Coordinate тип

Тип географические координаты предназначен для выбора координат на карте в формате долгота и широта. UI элементов будет простой – два Label и два TextField.

Представленная JSON-схема демонстрирует принципиальную гибкость, позволяя легко адаптировать систему под различные требования. Ее расширяемость проявляется в возможности добавления новых типов настроек простой реализацией соответствующих UI-компонентов без необходимости изменять базовую архитектуру. Дополнительные атрибуты для существующих типов, такие как шаг для числовых полей или ограничение длины для строк, могут вводиться естественным образом. Особое внимание уделено поддержке своих типов данных, что ярко проявляется в реализации составных параметров вроде координат.

Архитектура строго разделяет логику и представление, вынося декларативное описание всех параметров в JSON-конфигурацию, отделенную от кода реализации. Такой подход позволяет генерировать пользовательский интерфейс автоматически на основе метаданных, значительно упрощая поддержку системы. Важным преимуществом является независимость бизнес-логики от визуального представления - изменения в интерфейсе не требуют модификации основной логики работы с настройками.

Система эффективно работает со сложными структурами данных, включая вложенные массивы с разнотипными элементами и составные типы. Поддержка динамических вариантов выбора, которые могут изменяться в процессе работы приложения.

4.2. Основные классы в C++

4.2.1. Класс ISetting

Код данного класса представлен в листинге 1. Первым был реализован абстрактный класс, в котором описаны свойства, которыми должны обладать все настройки. ISetting служит единой точкой взаимодействия для всех типов параметров конфигурации, обеспечивая полиморфное поведение через чисто виртуальные методы. Это позволяет системе работать с любыми настройками, не зная их конкретной реализации.

Контракт данных. Четко определяет обязательные атрибуты любой настройки:

* Технический идентификатор (name)
* Отображаемое имя (caption)
* Описание (description)
* Значение по умолчанию
* Тип данных
* Флаг обязательности

Интеграция с JSON. Специальный метод default\_value() возвращает значение в формате, совместимом с JSON, что упрощает сериализацию/десериализацию конфигурации.

Аналогичный класс был создан для группы настроек ISettingsGroup

4.2.2. Класс ISelectionSetting

Код данного класса представлен в листинге 2. ISelectionSetting представляет собой специализированный интерфейс для настроек с предопределенными вариантами выбора

Основные методы:

* get\_selection\_list() - возвращает доступные варианты в JSON-формате
* is\_valid\_value() - проверяет соответствие значения допустимым вариантам

4.2.3. Класс IRegexStringSetting

Код данного класса представлен в листинге 3. Интерфейс предоставляет два основных метода:

* get\_regex\_pattern() – возвращает регулярное выражение
* is\_value\_match\_regex(const std::string &value) – вспомогательная определенная функция, которая позволяет проверить соответствует ли строка регулярному выражению

4.2.4. Прочие классы

Аналогично созданы классы ICoordinateSetting, IFileSetting

4.2.5. Фабричные классы ISettingParser ISettingsGroupParser ISettingsJsonParser

Код для данных классов представлен в листинге 6. Представленные интерфейсы реализуют паттерн проектирования "Абстрактная фабрика"[1].

Система организована по трехуровневому принципу:

1. ISettingParser  
   Базовый фабричный метод для создания отдельных настроек

virtual std::unique\_ptr<ISetting> parse(const Json::Value& json) = 0;

1. ISettingsGroupParser  
   Фабрика групп настроек, которая:
   * Создает группы (ISettingsGroup)
   * Использует композицию через setting\_parser()

virtual std::shared\_ptr<ISettingParser> setting\_parser() const = 0;

1. ISettingsJsonParser  
   Фабрика верхнего уровня, которая:
   * Обрабатывает целые JSON-структуры
   * Делегирует создание групп через group\_parser()

virtual auto group\_parser() const -> std::shared\_ptr<ISettingsGroupParser> = 0;

Каждый парсер верхнего уровня содержит фабрику для создания компонентов нижнего уровня (через setting\_parser()/group\_parser()), что обеспечивает:

* Гибкость в подмене реализаций
* Отсутствие жестких зависимостей
* Возможность кастомизации на любом уровне

ISettingsJsonParser предлагает два варианта создания объектов:

parse(const std::string& json\_file) // Из файла

parse(const Json::Value& json) // Из готового JSON

Каждая фабрика отвечает только за один тип объектов:

* ISettingParser → ISetting
* ISettingsGroupParser → ISettingsGroup
* ISettingsJsonParser → ISettingsJson

Можно выделить следующие преимущества созданной архитектуры фабрик:

1. Гибкость. Можно создать разные реализации парсеров для:
   * Альтернативных форматов (XML, YAML)
   * Специфичных версий JSON-схем
   * Оптимизированных или отладочных версий
2. Тестируемость. Легко подменять реальные парсеры mock-объектами
3. Единая точка расширения. Добавление нового типа настроек требует:
   * Реализации ISetting
   * Создания соответствующего ISettingParser
   * Интеграции через существующие фабрики

4.2.6. Model класс TabsModel

Система Model-View[4] в Qt представляет собой мощный инструмент для разделения данных и их представления, что особенно актуально при разработке сложных пользовательских интерфейсов. В контексте библиотеки управления настройками, эта архитектура позволяет эффективно организовать работу с иерархическими конфигурационными данными, обеспечивая при этом гибкость и производительность.

Основная идея заключается в том, что модель отвечает за хранение и обработку данных, а представление занимается исключительно их отображением. Связь между ними осуществляется через механизм ролей и систему сигналов-слотов, что обеспечивает автоматическую синхронизацию при изменении данных.

Класс TabsModel представляет собой центральный компонент системы управления настройками, выполняющий критически важную роль связующего звена между бизнес-логикой приложения и пользовательским интерфейсом. Код для данного класса приведен в листинге 4.

TabsModel реализует концепцию иерархического представления настроек, организованных по принципу вкладок, где каждая вкладка соответствует логической группе параметров. Модель наследует QAbstractListModel, что позволяет ей интегрироваться с QML-интерфейсом, обеспечивая автоматическую синхронизацию данных между C++ и QML слоями приложения.

Архитектура класса построена на нескольких фундаментальных принципах:

1. Инкапсуляция сложности - несмотря на то, что модель взаимодействует с парсерами JSON, управляет коллекцией подмоделей и обрабатывает изменения данных, ее внешний интерфейс остается простым и понятным.
2. Реактивность - модель активно использует механизм сигналов и свойств Qt для мгновенного отражения изменений в пользовательском интерфейсе.
3. Гибкость - благодаря использованию абстракций (ISettingsJson) и стратегии внедрения зависимостей, модель может работать с различными форматами конфигураций и системами парсинга.

Одной из наиболее значимых особенностей TabsModel является ее двухуровневая структура данных. Верхний уровень представляет собой список групп настроек (вкладок), в то время как каждый элемент этого списка содержит собственную SettingsListModel с конкретными параметрами. Такая организация обеспечивает:

* Модульность - возможность независимого обновления отдельных групп
* Масштабируемость - простоту добавления новых категорий настроек
* Производительность - минимизацию перерисовок интерфейса при изменениях

Установка входного файла через setInputFile запускает цепочку операций:

* Парсинг JSON-конфигурации
* Создание соответствующей структуры подмоделей
* Инициализация значений по умолчанию

Последующие изменения параметров автоматически отмечают модель как измененную (m\_has\_changes), что обеспечивает контроль целостности данных.

Операции сохранения (saveToFile) и восстановления (sourceFromOutputFile) реализованы с учетом необходимости сохранения формата и структуры исходной конфигурации.

Для QML-разработчика TabsModel предоставляет интуитивно понятный интерфейс, организованный вокруг трех ключевых ролей:

* TabNameRole - идентификатор группы
* TabCaptionRole - отображаемое название
* TabListModelRole - вложенная модель параметров

4.2.7. Model класс SettingsListModel

Данный класс используется для отображения группы настроек в QML по аналогии с предыдущим классом используя систему ролей и QAbstractListModel. Ключевые особенности класса заключаются в обширном использовании полиморфизма, а также в гибкой системе ролей. Код данного класса располагается в листинге 5.

Для работы с введенными значениями были созданы два абстрактных класса – IsettingValue и ISettingValueFactory.

4.2.8. Класс ISettingValue

Код данного класса представлен в листинге 7. Интерфейс ISettingValue служит строго типизированным контейнером для значений параметров конфигурации, обеспечивая контроль целостности данных в системе. Его ключевая задача — гарантировать соответствие между устанавливаемыми значениями и типом конкретной настройки.

4.2.9. Класс SimpleGetSetValueMixin

Код данного класса представлен в листинге 8. Класс SimpleGetSetValueMixin представляет собой шаблонный класс «примесь», разработанный для устранения дублирования кода при работе с простыми типами настроек. Его основное назначение — предоставить базовую реализацию интерфейса ISettingValue для элементарных типов данных, таких как целые числа, числа с плавающей точкой, булевы значения и другие аналогичные типы.

Класс выполняет функцию промежуточного слоя между абстрактным интерфейсом ISettingValue и конкретными реализациями для различных типов данных. Такой подход позволяет:

1. Централизовать общую логику работы с простыми значениями
2. Минимизировать дублирование кода
3. Обеспечить единообразие поведения для всех элементарных типов
4. Упростить добавление новых простых типов в будущем

Шаблон использует два параметра:

* JsonType\_t — конкретный класс-наследник, который будет использовать функциональность «примеси»
* SettingType\_v — тип настройки, с которым работает данный класс

Такой дизайн позволяет создавать специализированные классы для каждого типа данных, наследуя при этом общую логику. Например, класс IntegerValue может быть реализован как наследник SimpleGetSetValueMixin<IntegerValue, SettingType::Integer>.

Класс предоставляет два фабричных метода create и create\_with\_given\_value, которые:

* Создают экземпляры классов-наследников
* Инициализируют их значениями по умолчанию или переданными значениями
* Выполняют базовую валидацию типов

Методы get\_value и set\_value реализуют:

* Хранение текущего значения (m\_value)
* Проверку типов при установке новых значений
* Обработку null-значений с учетом флага обязательности настройки

Дополнительно класс хранит ссылку на объект ISetting (m\_setting), что позволяет:

* Получать информацию о типе и ограничениях
* Доступ к значению по умолчанию
* Проверять обязательность настройки

4.3. Основные QML элементы

4.3.1 SettingsEditor.qml

Файл SettingsEditor.qml представляет собой корневой визуальный элемент библиотеки, выполняющий ключевую координирующую роль в пользовательском интерфейсе. Этот компонент организует общую структуру окна настроек, включая навигацию по вкладкам и область отображения параметров. Базовым элементом для этого файла является Pane, что позволяет потом помещать это окно настроек в диалоговые окна, в полноценные отдельные окна, в свои UI компоненты. Такой способ компоновки элементов делает SettingsEditor очень гибким элементом.

Основная концепция построения SettingsEditor основана на разделении интерфейса на две логические части:

1. Панель вкладок (TabBar) - обеспечивает навигацию между группами настроек
2. Область содержимого (StackLayout) - динамически отображает параметры выбранной группы

Особенностью реализации является использование модели TabsModel в качестве единого источника данных для обеих частей интерфейса, что гарантирует согласованность состояния и минимизирует возможные расхождения.

API компонента SettingsEditor выглядит следующим образом:

* Поле sourceFile – путь к JSON файлу с описанием структуры настроек
* Поле outputFile – желаемый путь для сохранения настроек
* Поле boldSettings – флаг для включения или отключения жирного шрифта имён параметров
* Поле showSaveButton – флаг для включения или отключения кнопки сохранения настроек
* Поле showCloseButton – флаг для включения или отключения кнопки закрытия
* Поле showResetToDefaultButton – флаг для включения или отключения кнопки сброса настроек до состояния по умолчанию
* Поле showReloadButton – флаг для включения или отключения кнопки обновления значений настроек из выходного json файла
* Поле hasUnsavedChanges – флаг только для чтения, отображающий несохраненные изменения
* Сигнал aboutToSave() – вызывается непосредственно перед сохранением настроек в файл
* Сигнал saved() – вызывается сразу после успешной записи текущих значений в выходной файл
* Сигнал close() – вызывается при нажатии кнопки Закрыть
* Функция save() – позволяет вручную сохранить имеющиеся настройки
* Функция resetToDefault() – позволяет сбросить все значения до состояния по умолчанию
* Функция setGroupHidden() – позволяет скрыть какую либо вкладку
* Функция sourceFromOutputFile() – позволяет загрузить текущие значения из выходного файла



Рисунок 11 - пример окна настроек

4.3.1 SettingsTable.qml

Компонент SettingsTable реализует детализированное отображение отдельных групп настроек. Его архитектура построена вокруг нескольких ключевых принципов:

1. Гибкость макета - использование ColumnLayout и RowLayout позволяет автоматически адаптировать интерфейс под:
   * Различную длину названий параметров
   * Разнородные типы элементов управления
   * Изменяемые размеры окна приложения
2. Динамическое создание интерфейса - благодаря Repeater и делегатной системе, компонент автоматически генерирует:
   * Заголовки параметров с поддержкой переноса текста
   * Соответствующие элементы управления для каждого типа данных
   * Разделители между настройками для улучшения читаемости
3. Интерактивные элементы - реализованные функции включают:
   * Всплывающие подсказки с подробными описаниями
   * Адаптивные элементы управления
   * Поддержку различных состояний (только чтение, обязательные поля)

Связь между SettingsEditor и SettingsTable организована через систему моделей и свойств, что обеспечивает:

1. Односторонний поток данных - изменения в модели автоматически обновляют интерфейс через механизм привязок QML
2. Двустороннюю синхронизацию - пользовательский ввод передается обратно в модель через унифицированный интерфейс
3. Согласованное состояние - текущая вкладка, видимость групп и другие параметры синхронизированы между компонентами

Реализованная структура компонентов предоставляет несколько существенных преимуществ:

1. Модульность - каждый компонент имеет четко определенную зону ответственности
2. Масштабируемость - добавление новых типов параметров не требует изменения базовой структуры
3. Производительность - динамическая загрузка делегатов оптимизирует использование ресурсов
4. Кастомизируемость - дизайн отдельных элементов можно изменять без влияния на общую логику

4.3.2. DelegateManager.qml

Очень важным требованием к приложению является собственный делегат для каждого типа настроек. Например, для типа данных boolean это должен быть Checkbox, для типа данных integer – поле с регулярным выражением, не позволяющее ввести некорректное значение и тд…

Qml относится к интерпретируемым языкам, это позволяет очень удобно выбирать и загружать нужный делегат именно во время выполнения программы. Это является аналогом полиморфизма в статических языках.

Корневым элементом DelegateManager.qml является Item, который предоставляет функцию getDelegate() и компоненты для всех возможных типов настроек. Таким образом, DelegateManager реализует паттерн «Фабрика делегатов».

4.3.3. SettingsEditorDialog.qml и SettingsEditorWindow.qml

Поскольку SettingsEditor не является полноценным самостоятельным компонентом, то были реализованы еще два qml компонента, которые можно уже полноценно использовать в других проектах, не задумываясь о внутреннем устройстве системы. Главное отличие этих двух компонентов от корневого SettingsEditor в том, что они обрабатывают сигналы close() и следят за несохраненными изменениями.

SettingsEditorDialog представляет собой qml компонент Dialog и его можно использовать если нет необходимости в создании отдельного окна. То есть, например, его можно будет поместить во всплывающее окно или другие подобные элементы.

SettingsEditorWindow является полноценным окном настроек, который также как и диалоговое окно обрабатывает событие кнопки «закрыть», проверяя наличие несохраненных изменений и дополняя при этом корневой элемент SettingsEditor.

Заключение

Разработанная система представляет собой сбалансированное решение, органично сочетающее строгую типизацию backend-логики с гибкостью динамического интерфейса. Архитектура построена на принципах модульности и четкого разделения ответственности между компонентами, что обеспечивает как стабильность работы, так и простоту расширения функциональности.

Использование реактивного подхода к управлению состоянием гарантирует согласованность данных между всеми слоями системы. Многоуровневая организация компонентов позволяет эффективно масштабировать решение для работы с любым объемом параметров конфигурации, сохраняя при этом интуитивность взаимодействия.

Архитектура успешно решает ключевую задачу - предоставляет удобный инструмент управления настройками, который легко интегрируется в существующие Qt-проекты и адаптируется под конкретные требования, оставаясь при этом надежным и производительным.

ГЛАВА 5. АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Разработанная библиотека управления настройками успешно завершила свой жизненный цикл разработки, продемонстрировав высокую практическую ценность и эффективность предложенных архитектурных решений. На сегодняшний день решение активно используется в нескольких проектах нашей компании, что служит лучшим подтверждением его надежности и удобства интеграции.

Ключевым достижением стало успешное размещение библиотеки в корпоративном Artifactory, что стандартизировало процесс ее подключения в новые проекты. Это позволило значительно упростить процедуру внедрения для различных команд разработки, обеспечивая:

* централизованное управление версиями
* автоматическое разрешение зависимостей
* воспроизводимость сборок

Практическое применение библиотеки в реальных проектах подтвердило правильность выбранных архитектурных принципов. Особенно ценными оказались:

* единый формат описания настроек
* автоматическая генерация интерфейса
* строгая типизация параметров

Развертывание решения в production-среде продемонстрировало его стабильность и производительность даже при работе с большими объемами конфигурационных данных. Успешный опыт интеграции в различные проекты компании свидетельствует о хорошей адаптируемости библиотеки к разным бизнес-задачам и техническим требованиям.

Библиотека стала важной частью инфраструктуры компании, сократив время разработки новых функциональных возможностей, связанных с управлением настройками, и повысив согласованность пользовательских интерфейсов в различных продуктах. Ее включение в корпоративный реестр компонентов открывает перспективы для дальнейшего развития и совершенствования решения.

ГЛАВА 6. ПЕРСПЕКТИВЫ РАЗВИТИЯ

Одним из важных возможных расширений библиотеки является реализация механизма валидации зависимостей между настройками, который позволит определять сложные бизнес-правила взаимодействия параметров. Особое внимание будет уделено разработке системы версионирования конфигураций, обеспечивающей плавную миграцию между разными форматами настроек при обновлениях продукта.

Другим возможным направлением развития может стать расширение возможностей кастомизации интерфейса, включая поддержку тем оформления (темный/светлый режим) и более гибкую систему визуальных стилей. Планируется внедрение механизма групповых операций с настройками, такого как массовое применение изменений или сброс группы параметров к значениям по умолчанию.

Очень важным потенциальным функционалом является система интеллектуального поиска по настройкам с возможностью фильтрации по различным критериям.

Техническое развитие библиотеки может включать оптимизацию работы с большими объемами конфигурационных данных, внедрение ленивой загрузки параметров и улучшение механизма кэширования. Особое внимание будет уделено расширению документации и созданию набора «примеры» для ускорения процесса интеграции в новые проекты компании.

ЗАКЛЮЧЕНИЕ

В рамках данной дипломной работы была успешно разработана и внедрена библиотека для управления настройками в Qt/QML-приложениях, которая прошла полный цикл от проектирования до промышленной эксплуатации. Реализованное решение продемонстрировало свою эффективность, став стандартом для работы с конфигурациями в нескольких проектах компании.

Ключевым достижением работы стало создание целостной архитектуры, сочетающей строгую типизацию C++ backend'а с гибкостью динамического QML-интерфейса. Разработанная система обладает рядом неоспоримых преимуществ:

* Единый стандарт описания настроек в JSON-формате
* Автоматическая генерация пользовательского интерфейса
* Надежная система валидации и контроля данных
* Простота интеграции в новые проекты

Особую ценность представляет успешное внедрение библиотеки в корпоративную экосистему, включая ее публикацию в локальном Artifactory, что стандартизировало процесс использования решения различными командами разработчиков. Практическое применение в реальных проектах подтвердило соответствие библиотеки требованиям производительности, надежности и удобства использования.

Проведенная работа не только решила актуальную техническую задачу, но и создала прочный фундамент для дальнейшего развития системы. Реализованная архитектура открывает широкие возможности для расширения функциональности, что делает библиотеку перспективным инструментом для будущих проектов компании.

Разработанное решение представляет собой законченный продукт, сочетающий в себе академическую строгость подхода с практической ориентированностью на реальные бизнес-задачи. Опыт создания и внедрения этой библиотеки стал ценным вкладом в совершенствование процессов разработки программного обеспечения в компании.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. "Приёмы объектно-ориентированного проектирования. Па́ттерны проектирования" / Эрих Гамма, Ричард Хелм, Джон Влисидис, Ральф Джонсон – 1994. – 395 с.
2. "Эффективный и современный С++" / Скотт Майерс – 2014. – 334 с.
3. "Mastering Conan" / Sergio R. Gomez – 2023. – 356 с.
4. "C++ GUI Programming With Qt 4" / Марк Саммерфилд, Ясмин Бланшетт – 2006. – 734 с.
5. Официальная документация коллекции компиляторов GNU // GCC online documentation. – 2025. – URL: <https://gcc.gnu.org/onlinedocs/> (дата обращения 12.05.2025)
6. Официальная документация пакетного менеджера conan // Conan 2 - C and C++ Package Manager Documentation. – 2020. – URL: <https://docs.conan.io/> (дата обращения 12.05.2025)
7. Центральный репозиторий пакетного менеджера конан // The Conan libraries and tools central repository. – 2020. – URL: <https://conan.io/center> (дата обращения 12.05.2025)
8. Документация библиотеки для логирование spdlog // Spdlog quickstart library documentation. – 2023-2024. – URL: <https://github.com/gabime/spdlog/wiki> (дата обращения 12.05.2025)
9. Официальная документация библиотеки fmt // Fmt library get started page. – 2025. – URL: <https://fmt.dev/11.1/get-started/> (дата обращения 12.05.2025)
10. Официальная документация Qt5 // Qt Documentation. – 2025. – URL: <https://doc.qt.io/qt-5/> (дата обращения 12.05.2025)
11. Официальная документация KDE Framework // KDE Documentation. – 2005. – URL: <https://docs.kde.org> (дата обращения 12.05.2025)

ПРИЛОЖЕНИЕ

Листинг 1 – код абстрактного класса ISetting

#pragma once

#include <string>

#include <memory>

#include <json/json.h>

#include <qml-settings-editor/settings/interface/constants/setting\_type.h>

#include <qml-settings-editor/settings/interface/types/i\_extended\_json\_value.h>

namespace qml\_settings\_editor::settings::interface::types {

*/// @brief Interface class created so that we could effectively replace settings in the future*

*/// @details This is base class for ALL settings*

  class ISetting

  {

   public:

    virtual ~ISetting() = default;

*/\*\**

*\* Get name as it should be saved in json*

*\*/*

    virtual auto get\_name() const -> const std::string& = 0;

*/\*\**

*\* Get name as it should be shown in UI*

*\* @note If caption is not set, return ""*

*\*/*

    virtual auto get\_caption() const -> const std::string& = 0;

*/\*\**

*\* @note If description is not set, return ""*

*\*/*

    virtual auto get\_description() const -> const std::string& = 0;

*/\*\**

*\* @note If default value is not set, return json null(NOT nullptr)*

*\*/*

    virtual auto default\_value() const -> std::shared\_ptr<const types::IExtendedJsonValue> = 0;

*/\*\**

*\* @return type of setting*

*\*/*

    virtual auto type() const -> constants::SettingType = 0;

*/\*\**

*\* @return true if setting is required*

*\*/*

    virtual auto is\_required() const -> bool = 0;

*/\*\**

*\* Get name as it should be shown in UI*

*\* @note If caption is not set, return name*

*\*/*

    virtual auto get\_caption\_or\_name() const -> const std::string&;

  };

} *// namespace qml\_settings\_editor::settings::interface*

Листинг 2 – абстрактный класс ISelectionSetting

#pragma once

#include <string>

#include <memory>

#include <json/json.h>

#include <qml-settings-editor/settings/interface/constants/setting\_type.h>

#include <qml-settings-editor/settings/interface/types/i\_extended\_json\_value.h>

namespace qml\_settings\_editor::settings::interface::types {

*/// @brief Interface represents setting with selectable options*

  class ISelectionSetting

  {

   public:

    virtual ~ISelectionSetting() = default;

*/\*\**

*\* @return List of available options*

*\* @note Return value should be json array*

*\*/*

    virtual auto get\_selection\_list() const -> const Json::Value& = 0;

*/\*\**

*\* @return true if value is in selection list. false otherwise*

*\*/*

    virtual bool is\_valid\_value(std::shared\_ptr<const types::IExtendedJsonValue> *value*) const

    {

      auto list = get\_selection\_list();

      return std::find(list.begin(), list.end(), value->value()) != list.end();

    }

  };

} *// namespace qml\_settings\_editor::settings::interface::types*

Листинг 3 – абстрактный класс IRegexStringSetting

#pragma once

#include <string>

#include <json/json.h>

#include <qml-settings-editor/settings/interface/constants/setting\_type.h>

#include <qml-settings-editor/settings/interface/types/i\_extended\_json\_value.h>

namespace qml\_settings\_editor::settings::interface::types {

*/// @brief Interface represents setting with selectable options*

  class IRegexStringSetting

  {

   public:

    virtual ~IRegexStringSetting() = default;

    virtual const std::string &get\_regex\_pattern() const = 0;

*// @return true if given string is valid for regex*

    virtual bool is\_value\_match\_regex(const std::string &*value*) const;

  };

} *// namespace qml\_settings\_editor::settings::interface::types*

Листинг 4 – Model-View класс TabsModel

#pragma once

#include <QObject>

#include <memory>

#include <vector>

#include <functional>

#include <utility>

#include <QAbstractListModel>

#include <json/json.h>

#include <qml-settings-editor/settings/interface/types/i\_settings\_json.h>

#include <qml-settings-editor/gui/settings\_list\_model/settings\_list\_model.h>

#include <qml-settings-editor/gui/settings\_list\_model/utils.h>

#include <qml-settings-editor/gui/settings\_list\_model/default\_impl/setting\_value\_factory.h>

namespace qml\_settings\_editor::gui::tabs\_model {

*/\*\**

*\* @details This class will provide all needed information for each tab in the setting window*

*\*          and will be used in the QML side*

*\*/*

  class TabsModel : public QAbstractListModel

  {

    Q\_OBJECT

    Q\_PROPERTY(QString *fileWithSettingsDescription* READ getInputFile WRITE setInputFile NOTIFY *inputFileChanged*)

    Q\_PROPERTY(QString outputFile READ getOutputFile WRITE setOutputFile NOTIFY outputFileChanged)

    Q\_PROPERTY(bool hasUnsavedChanges READ has\_unsaved\_changes NOTIFY hasUnsavedChangesChanged)

   public:

    explicit TabsModel(QObject\* parent = nullptr);

    ~TabsModel() override = default;

    enum class Roles

    {

      TabNameRole = Qt::UserRole + 1, */// @brief Basically it will be group name*

      TabCaptionRole, */// @brief Basically it will be group caption*

      TabListModelRole                */// @brief Basically it will be settings from group*

    };

    Q\_INVOKABLE QVariant data(const QModelIndex& index, int role) const override;

    Q\_INVOKABLE int rowCount(const QModelIndex& *parent* = {}) const override;

    Q\_INVOKABLE QHash<int, QByteArray> roleNames() const override;

    Q\_INVOKABLE const QString& getInputFile() const;

*/// @brief After we set input file we will have settings groups*

*/// and model will become actually filled with values*

    Q\_INVOKABLE void setInputFile(QString *input\_file*);

    Q\_INVOKABLE const QString& getOutputFile() const;

*/// @brief After we set output file we will be able to save out values*

    Q\_INVOKABLE void setOutputFile(QString *output\_file*);

    Q\_INVOKABLE void setGroupHidden(const QString& *group\_name*, bool *is\_hidden*);

    Q\_INVOKABLE void saveToFile() const;

    Q\_INVOKABLE void sourceFromOutputFile();

    Q\_INVOKABLE void resetToDefault();

   signals:

    void inputFileChanged(QString *input\_file*);

    void outputFileChanged(QString *output\_file*);

    void hasUnsavedChangesChanged();

   private:

    QString m\_input\_file;

    QString m\_output\_file;

*/// @brief This field will provide meta information about each setting*

*/// @note This field will be initialized when setInputFile is called*

    std::shared\_ptr<settings::interface::types::ISettingsJson> m\_settings\_json;

*/// @details This field will provide table model for each tab*

*/// @note This field will be initialized when setInputFile is called*

    std::vector<std::shared\_ptr<settings\_list\_model::SettingsListModel>> m\_tab\_tree\_models;

*/// @brief This functor is used to provide values from already existing config*

    std::shared\_ptr<settings\_list\_model::utils::ExistingConfigValuesProvider>

      m\_existing\_config\_values\_provider =

        std::make\_shared<settings\_list\_model::utils::ExistingConfigValuesProvider>("");

    std::shared\_ptr<settings\_list\_model::interface::ISettingValueFactory> m\_setting\_value\_factory =

      std::make\_shared<settings\_list\_model::default\_impl::SettingValueFactory>();

    mutable bool m\_has\_changes = false;

    bool has\_unsaved\_changes() const { return m\_has\_changes; }

    void set\_empty\_input\_file();

  };

} *// namespace qml\_settings\_editor::gui::tabs\_model*

Листинг 5 – класс SettingsListModel

#pragma once

#include <QObject>

#include <memory>

#include <functional>

#include <utility>

#include <QAbstractListModel>

#include <QMetaType>

#include <json/json.h>

#include <qml-settings-editor/settings/interface/types/i\_settings\_json.h>

#include <qml-settings-editor/gui/settings\_list\_model/interface/i\_setting\_value.h>

#include <qml-settings-editor/gui/settings\_list\_model/interface/i\_setting\_value\_factory.h>

#include <qml-settings-editor/gui/settings\_list\_model/utils.h>

namespace qml\_settings\_editor::gui::settings\_list\_model

{

  enum class SettingsColumns

  {

    NAME = 0,

    CAPTION,

    TYPE,

    VALUE,

    COUNT = 4

  };

*/// @brief This class represents settings model that will be used in the QML side*

*/// It takes settings from given group and provides them to the QML side*

  class SettingsListModel : public QAbstractListModel

  {

    Q\_OBJECT

    Q\_PROPERTY(bool *isHidden* READ is\_hidden WRITE set\_hidden NOTIFY *isHiddenChanged*)

   public:

    SettingsListModel() = default;

    SettingsListModel(

      std::shared\_ptr<settings::interface::types::ISettingsGroup> *settings\_group*,

      std::shared\_ptr<utils::ExistingConfigValuesProvider> *config\_values\_provider*,

      std::shared\_ptr<interface::ISettingValueFactory> *setting\_value\_factory*

    );

    virtual ~SettingsListModel() = default;

    enum class Roles

    {

      Name = Qt::UserRole + 1,

      Caption,

      Type,

      Value,

      Description,

*// Selection Setting roles*

      SelectionList,

      SelectionCurrentIndex,

*// Selection Setting roles*

*// FilePath Setting roles*

      FilePathExtensions,

*// FilePath Setting roles*

*// Array Setting roles*

      JsonTableModel,

*// Array Setting roles*

*// Regex String Setting roles*

      Regex,

*// Regex String Setting roles*

*// Coordinate Setting roles*

      CoordinateLatitude,

      CoordinateLongitude,

*// Coordinate Setting roles*

    };

    QHash<int, QByteArray> roleNames() const override;

    QVariant data(QModelIndex const& *index*, int *role*) const override;

    bool setData(QModelIndex const& *index*, QVariant const& *value*, int *role* = Qt::EditRole) override;

    Q\_INVOKABLE bool setData(int *row*, QVariant const& *value*);

    Qt::ItemFlags flags(QModelIndex const& *index*) const override;

    QVariant headerData(

      int *section*,

      Qt::Orientation *orientation*,

      int *role* = Qt::DisplayRole

    ) const override;

    int rowCount(QModelIndex const& *parent* = {}) const override;

    void create\_setting\_values(

      std::shared\_ptr<interface::ISettingValueFactory> const& *setting\_value\_factory*,

      std::shared\_ptr<utils::ExistingConfigValuesProvider> *config\_values\_provider*

    );

    void source\_values\_from\_output\_file(

      std::shared\_ptr<utils::ExistingConfigValuesProvider> *config\_values\_provider*

    );

    bool is\_setting\_required(int *row*) const

    {

      return m\_setting\_values.at(row)->get\_setting()->is\_required();

    }

    Json::Value get\_settings\_with\_values() const;

    bool is\_hidden() const { return m\_is\_hidden; }

    void set\_hidden(bool *value*) { m\_is\_hidden = *value*; emit isHiddenChanged(); }

    std::shared\_ptr<settings::interface::types::ISettingsGroup> get\_settings\_group() const

    {

      return m\_settings\_group;

    }

    void reset\_to\_default();

   signals:

    void isHiddenChanged();

    void changesCommited();

   private:

    std::shared\_ptr<settings::interface::types::ISettingsGroup> m\_settings\_group;

    std::vector<std::shared\_ptr<interface::ISettingValue>> m\_setting\_values;

    bool m\_is\_hidden = false;

  };

} *// namespace qml\_settings\_editor::gui::settings\_list\_model*

Q\_DECLARE\_METATYPE(qml\_settings\_editor::gui::settings\_list\_model::SettingsListModel\*)

Листинг 6 – ISettingParser, ISettingsGroupParser, ISettingsJsonParser

#pragma once

#include <memory>

#include <json/json.h>

#include <qml-settings-editor/settings/interface/types/i\_setting.h>

namespace qml\_settings\_editor::settings::interface::parsers

{

/// @brief Interface class created so that we could effectively replace parsers in the future

class ISettingParser

{

public:

virtual ~ISettingParser() = default;

virtual std::unique\_ptr<interface::types::ISetting> parse(const Json::Value & json) = 0;

};

}

#pragma once

#include <memory>

#include <json/json.h>

#include <qml-settings-editor/settings/interface/types/i\_settings\_group.h>

#include <qml-settings-editor/settings/interface/parsers/i\_setting\_parser.h>

namespace qml\_settings\_editor::settings::interface::parsers

{

/// @brief Interface class created so that we could effectively replace parsers in the future

/// Propogate system is used here to provide setting\_parser

class ISettingsGroupParser

{

public:

virtual ~ISettingsGroupParser() = default;

virtual std::unique\_ptr<interface::types::ISettingsGroup> parse(const Json::Value & json) = 0;

virtual std::shared\_ptr<ISettingParser> setting\_parser() const = 0;

};

}

#pragma once

#include <memory>

#include <qml-settings-editor/settings/interface/types/i\_settings\_json.h>

#include <qml-settings-editor/settings/interface/parsers/i\_settings\_group\_parser.h>

namespace qml\_settings\_editor::settings::interface::parsers

{

/// @brief Interface class created so that we could effectively replace parsers in the future

/// Propogate system is used here to provide group\_parser

class ISettingsJsonParser

{

public:

virtual ~ISettingsJsonParser() = default;

virtual auto parse(const std::string& json\_file)

-> std::unique\_ptr<settings::interface::types::ISettingsJson> = 0;

virtual auto parse(const Json::Value& json)

-> std::unique\_ptr<settings::interface::types::ISettingsJson> = 0;

virtual auto group\_parser() const -> std::shared\_ptr<settings::interface::parsers::ISettingsGroupParser> = 0;

};

} // namespace qml\_settings\_editor::settings::interface

Листинг 7 – интерфейсный класс ISettingValue

#pragma once

#include <qml-settings-editor/settings/interface/types/i\_setting.h>

namespace qml\_settings\_editor::gui::settings\_list\_model::interface {

/// @brief Interface for setting value

/// It is useful because it takes responsibility to make sure that given value is

/// compatible with the setting that it contains

class ISettingValue

{

public:

virtual ~ISettingValue() = default;

/// @return Current saved value for the setting

virtual auto get\_value() const -> std::shared\_ptr<settings::interface::types::IExtendedJsonValue> = 0;

/// @brief Function sets new value for the setting

/// @return True if value was successfully set. False if given value is not compatible with the setting

/// for any reason.

/// @example If we try to put "sus" in integer value - it will return false

virtual auto set\_value(const std::shared\_ptr<settings::interface::types::IExtendedJsonValue>& value) -> bool = 0;

/// @return Setting that this value belongs to

virtual auto get\_setting() const -> std::shared\_ptr<settings::interface::types::ISetting> = 0;

};

} // namespace qml\_settings\_editor::gui::settings\_list\_model::interface

Листинг 8 – класс SimpleGetSetValueMixin

#pragma once

#include <cassert>

#include <spdlog/spdlog.h>

#include <qml-settings-editor/gui/settings\_list\_model/interface/i\_setting\_value.h>

#include <qml-settings-editor/settings/default\_impl/types/extended\_json\_value.h>

#include <qml-settings-editor/settings/default\_impl/utils/utils.h>

namespace qml\_settings\_editor::gui::settings\_list\_model::default\_impl {

  using settings::default\_impl::utils::name\_from\_setting\_type;

*/// For some simple json types like integer, uinteger, double, bool and maybe some others*

*/// we simply need to check if given value type is convertible to setting type and thats all.*

*/// So in order to avoid huge code duplication this mixin class was designed.*

*/// This class will represent IntegerValue, UintegerValue, DoubleValue, BoolValue and maybe others*

*/// @tparam JsonType\_t it should be inherited class like IntegerValue*

*/// @tparam SettingType\_v type of setting this mixin will sustain. e.g. for IntegerValue it will*

*/// be SettingType::Integer*

  template <class JsonType\_t, settings::interface::constants::SettingType SettingType\_v>

  class SimpleGetSetValueMixin : public virtual interface::ISettingValue

  {

   public:

*/// @copydoc ISettingValueFactory::create(std::shared\_ptr<settings::interface::types::ISetting>*

*/// setting)*

    static std::unique\_ptr<JsonType\_t>

    create(std::shared\_ptr<settings::interface::types::ISetting> *setting*);

*/// @copydoc ISettingValueFactory::create\_with\_given\_value(std::shared\_ptr<ISetting> setting,*

*/// std::shared\_ptr<IExtendedJsonValue> value)*

    static std::unique\_ptr<JsonType\_t>

    create\_with\_given\_value(std::shared\_ptr<settings::interface::types::ISetting> *setting*,

                            std::shared\_ptr<settings::interface::types::IExtendedJsonValue> *value*);

*/// @copydoc ISettingValue::get\_value()*

    auto get\_value() const

      -> std::shared\_ptr<settings::interface::types::IExtendedJsonValue> override final;

*/// @copydoc ISettingValue::set\_value(const*

*/// std::shared\_ptr<settings::interface::types::IExtendedJsonValue>& value)*

    auto set\_value(const std::shared\_ptr<settings::interface::types::IExtendedJsonValue>& *value*)

      -> bool override final;

*/// @copydoc ISettingValue::get\_setting()*

    auto get\_setting() const

      -> std::shared\_ptr<settings::interface::types::ISetting> override final;

   protected:

    std::shared\_ptr<settings::interface::types::ISetting> m\_setting;

    std::shared\_ptr<settings::interface::types::IExtendedJsonValue> m\_value;

  };

  template <class JsonType\_t, settings::interface::constants::SettingType SettingType\_v>

  std::unique\_ptr<JsonType\_t> SimpleGetSetValueMixin<JsonType\_t, SettingType\_v>::create(

    std::shared\_ptr<settings::interface::types::ISetting> *setting*)

  {

*// It is job of a parser to check that setting type is the same as in mixin*

    assert(SettingType\_v == setting->type() && "Setting type mismatch");

    auto result = std::unique\_ptr<JsonType\_t>(new JsonType\_t());

    result->m\_value = setting->default\_value()->clone();

    result->m\_setting = std::move(setting);

    return result;

  }

  template <class JsonType\_t, settings::interface::constants::SettingType SettingType\_v>

  std::unique\_ptr<JsonType\_t>

  SimpleGetSetValueMixin<JsonType\_t, SettingType\_v>::create\_with\_given\_value(

    std::shared\_ptr<settings::interface::types::ISetting> setting,

    std::shared\_ptr<settings::interface::types::IExtendedJsonValue> value)

  {

*// It is job of a parser to check that setting type is the same as in mixin*

    assert(SettingType\_v == setting->type() && "Setting type mismatch");

    auto result = std::unique\_ptr<JsonType\_t>(new JsonType\_t());

*// ^ If given value type is not the same as in default value, use default value*

*// ^ The only exception is if value is null and setting is not required*

    if (not value->is\_convertible\_to(setting->default\_value()->type()) and

        not(value->is\_null() and not setting->is\_required())) {

      spdlog::debug(

        "Value for setting {} has wrong type. Expected {}. Using default value instead.",

        setting->get\_name(), name\_from\_setting\_type(setting->default\_value()->type()));

      result->m\_value = setting->default\_value()->clone();

    } else {

      result->m\_value = value;

    }

    result->m\_setting = std::move(setting);

    return result;

  }

  template <class JsonType\_t, settings::interface::constants::SettingType SettingType\_v>

  auto SimpleGetSetValueMixin<JsonType\_t, SettingType\_v>::get\_value() const

    -> std::shared\_ptr<settings::interface::types::IExtendedJsonValue>

  {

    return m\_value;

  }

  template <class JsonType\_t, settings::interface::constants::SettingType SettingType\_v>

  auto SimpleGetSetValueMixin<JsonType\_t, SettingType\_v>::set\_value(

    const std::shared\_ptr<settings::interface::types::IExtendedJsonValue>& *new\_value*) -> bool

  {

*// Check if someone tries to put null in setting when one is required*

    if (*new\_value*->is\_null()) {

      if (m\_setting->is\_required()) {

        spdlog::debug("Value for setting {} cannot be null.", m\_setting->get\_name());

        return false;

      }

      m\_value = new\_value;

      return true;

    }

*// Check if someone tries to put incorrect type in setting*

    if (not new\_value->is\_convertible\_to(m\_setting->type())) {

      spdlog::debug("Value for setting {} has wrong type. Expected {}.",

                    name\_from\_setting\_type(new\_value->type()),

                    name\_from\_setting\_type(m\_setting->default\_value()->type()));

      return false;

    }

    m\_value = new\_value;

    return true;

  }

  template <class JsonType\_t, settings::interface::constants::SettingType SettingType\_v>

  auto SimpleGetSetValueMixin<JsonType\_t, SettingType\_v>::get\_setting() const

    -> std::shared\_ptr<settings::interface::types::ISetting>

  {

    return m\_setting;

  }

} *// namespace qml\_settings\_editor::gui::settings\_list\_model::default\_impl*

Листинг 9 – SettingsEditor.qml

Pane {

    id: pane

    property string sourceFile: ""

    property string outputFile: ""

    property bool boldSettings: false

    property bool showSaveButton: true

    property bool showCloseButton: true

    property bool showResetToDefaultButton: true

    property bool showReloadButton: true

    property bool hasUnsavedChanges: tabsModel.hasUnsavedChanges

    signal aboutToSave()

    signal saved()

    signal close()

    ColumnLayout {

        anchors.fill: parent

        TabBar {

            id: tabBar

            clip: true

            Layout.fillWidth: true

            Component.onCompleted: {

               ...

            }

            Repeater {

                model: tabsModel

                TabButton {

                    ...

                }

            }

        }

        StackLayout {

            currentIndex: tabBar.currentIndex

            clip: true

            Repeater {

                model: tabsModel

                SettingsTable {

                    id: table

                    settingsListModel: model.tabListModel

                }

            }

        }

        RowLayout {

            Button {

                id: resetToDefault

                text: "По умолчанию"

            }

            Button {

                id: reload

                text: "Перезагрузить"

                visible: pane.showReloadButton

                onClicked: {

                    tabsModel.sourceFromOutputFile();

                }

            }

            Item {

                Layout.fillWidth: true

            }

            Button {

                id: saveButton

                text: "Сохранить"

            }

            Button {

                id: closeButton

                text: "Закрыть"

            }

        }

    }

    TabsModel {

        id: tabsModel

        outputFile: pane.outputFile

        fileWithSettingsDescription: sourceFile

    }

}