СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 1](#_Toc197726092)

[ВВЕДЕНИЕ 2](#_Toc197726093)

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 3](#_Toc197726094)

[1.1. Обоснование актуальности работы 3](#_Toc197726095)

[1.2. Обзор существующих решений 3](#_Toc197726096)

[1.3. Анализ существующих решений 4](#_Toc197726097)

[QSettings (Qt Core) 5](#_Toc197726098)

[Qt.labs.settings (QML) 5](#_Toc197726099)

[KConfig (KDE Framework) 5](#_Toc197726100)

[1.4 Заключение 6](#_Toc197726101)

[ГЛАВА 3. ВЫБОР СПОСОБА ПОСТАВКИ БИБЛИОТЕКИ 7](#_Toc197726102)

[3.1. Установка системной зависимости 7](#_Toc197726103)

[3.2. Использовать git submodule 10](#_Toc197726104)

[3.3. Использовать пакетный менеджер conan 13](#_Toc197726105)

[ГЛАВА 2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ 17](#_Toc197726106)

[2.1. Функциональные требования 17](#_Toc197726107)

[2.2. Нефункциональные требования 17](#_Toc197726108)

[ГЛАВА 3. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ 17](#_Toc197726109)

[ЗАКЛЮЧЕНИЕ 17](#_Toc197726110)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 17](#_Toc197726111)

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.

1.4 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 – библиотека для удобного форматирования строк
* Spdlog – 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.

3.1. 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.

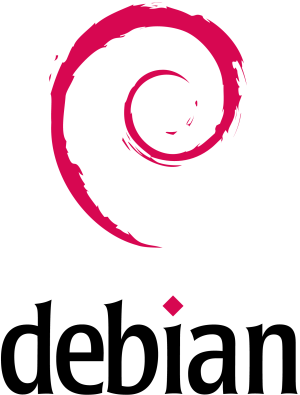


Рисунок 1 - логотип Debian

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.

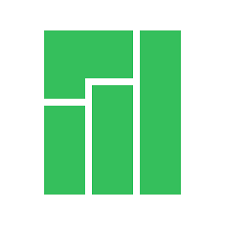


Рисунок 2 - логотип Manjaro

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.

3.2. 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.

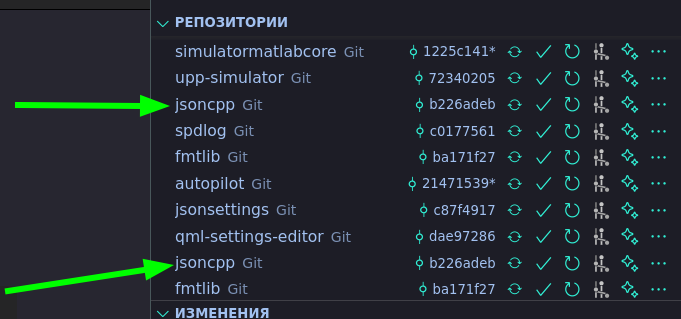


Рисунок 4 - проблема разных версий библиотек

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.

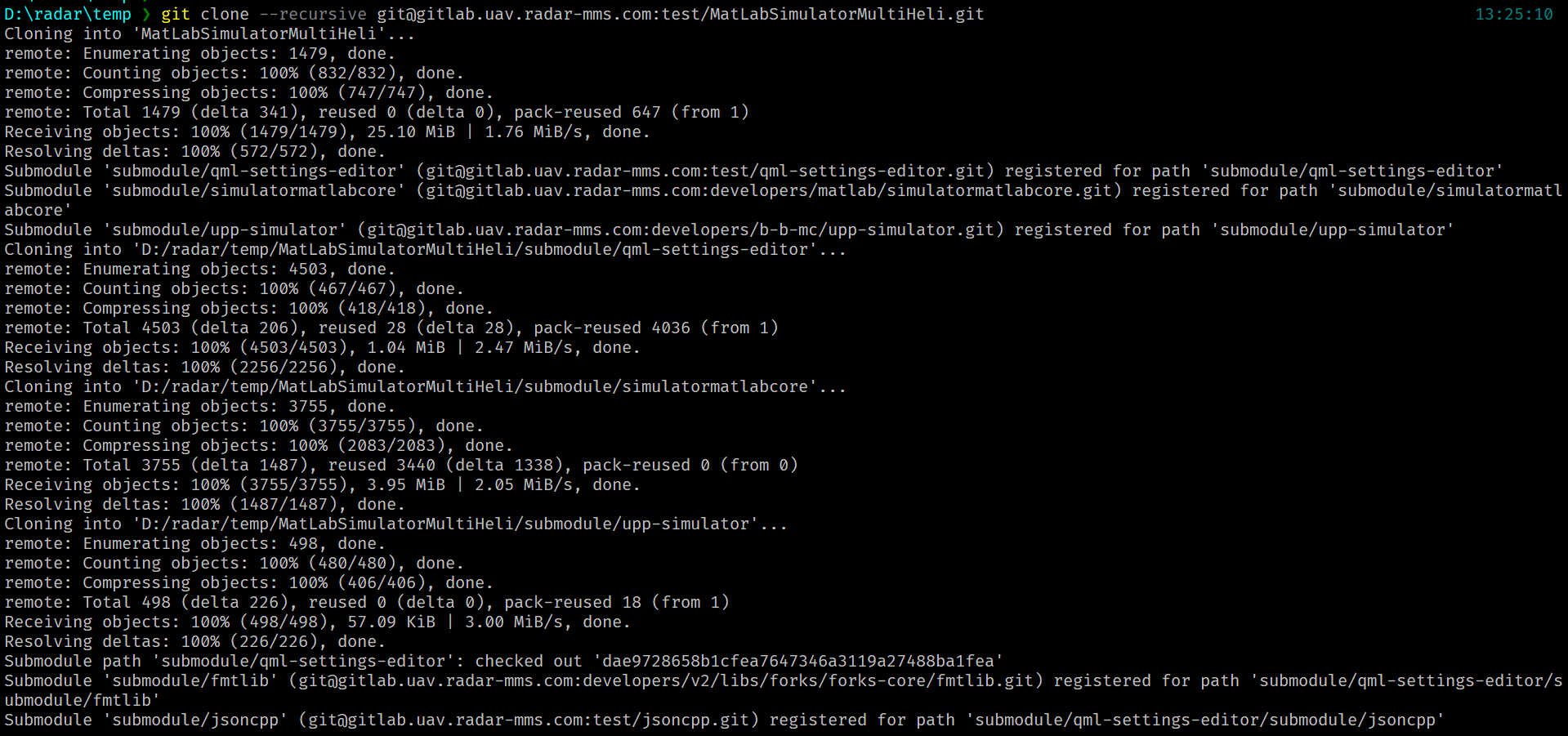


Рисунок 5 - пример клонирования проекта с сабмодулями

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.

3.3. Использовать пакетный менеджер conan

В контексте данной работы использование системы управления пакетами Conan было выбрано в качестве основного инструмента для решения этих задач.



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.

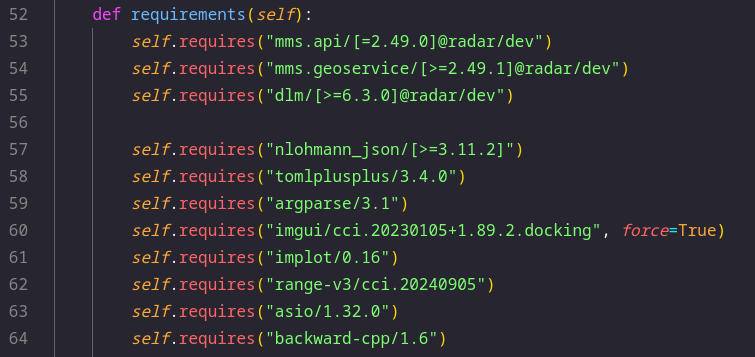


Рисунок 7 - пример conanfile.py

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.

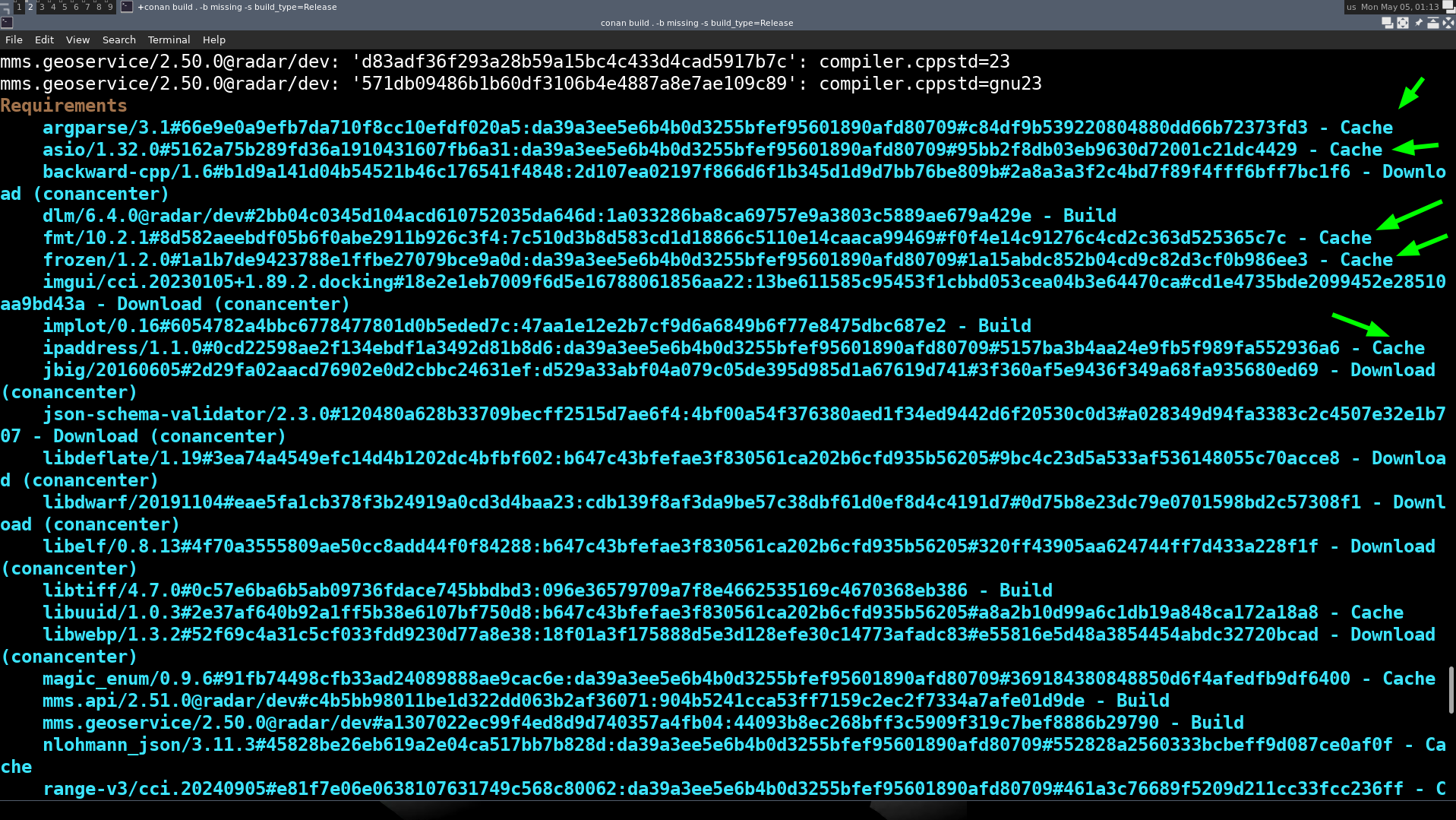


Рисунок 8 - прпимер кеширования библиотек

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.

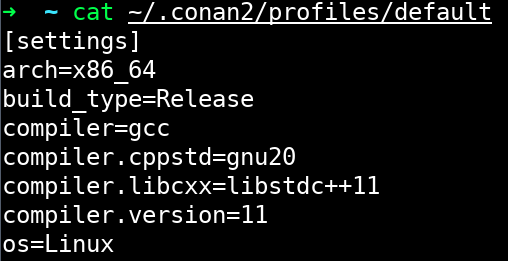


Рисунок 9 - пример профиля conan для обычной сборки проекта

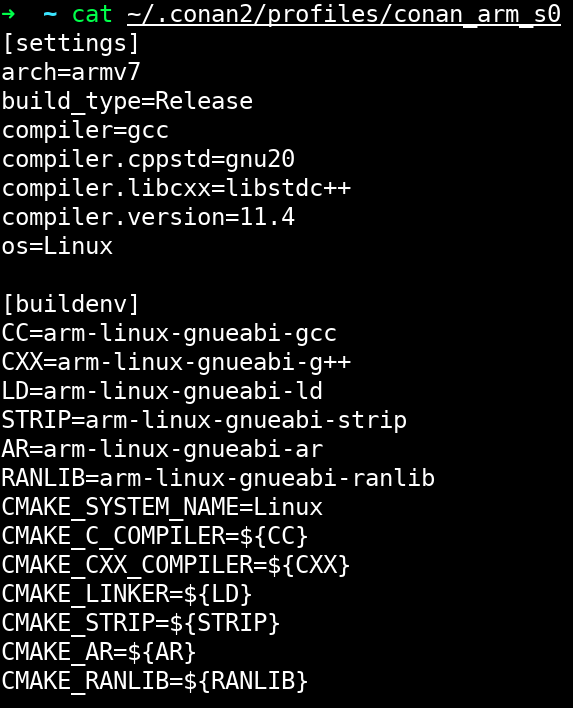


Рисунок 10 - пример профиля conan для кросс компиляции всех библиотек под arm

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 большим, и чтобы не пришлось переписывать его много раз, необходимо учесть все важные детали разработки и реализации. Традиционно требования разделяют на функциональные и нефункциональные.

2.1. 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)

2.2. 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 3. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ

3.1. Структура JSON

Для начала необходимо разобраться со структурой json. Корневой элемент можно сделать следующим образом:

{

"groups": [...]

}

Каждая группа будет состоять из следующих полей:

{

"name": "block\_1",

"caption": "Test Block Name 1",

"settings": [...]

}

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

Каждая настройка будет выглядеть следующим образом:

{

"name": "unique\_id",

"caption": "Отображаемое имя",

"description": "Подсказка",

"type": "Тип",

"default\_value": ...,

"is\_required": false

}

Для настроек доступны следующие типы

3.1.1. Boolean тип

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

3.1.2. Integer/Real/UInt типы

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

3.1.3. String тип

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

3.1.4. 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.

3.1.5. 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

3.1.6. Selection тип

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

Json описания этой настройки имеет дополнительное поле options

3.1.7. 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.

Анализируя полученное решение, можно выделить следующие преимущества предложенного решения

1. Универсальность и расширяемость

Представленная JSON-схема обладает принципиальной гибкостью, позволяя:

* Добавлять новые типы настроек без изменения базовой архитектуры (достаточно реализовать соответствующий UI-компонент)
* Вводить дополнительные атрибуты для существующих типов (например, step для числовых полей или max\_length для строк)
* Поддерживать кастомные типы данных (как в случае с Coordinate)

2. Четкое разделение логики и представления

* Декларативное описание всех параметров в JSON отделено от кода реализации
* Автоматическая генерация UI на основе метаданных упрощает поддержку
* Изменение интерфейса не требует модификации бизнес-логики

3. Поддержка сложных структур данных

* Вложенные массивы (Array) с гетерогенными элементами
* Составные типы (Coordinate как массив из двух чисел)
* Динамические варианты выбора (Selection)

4. Готовность к интернационализации

* Поля caption и description обеспечивают:
  + Локализацию через внешние translation-файлы
  + Контекстные подсказки для пользователей
  + Возможность динамического обновления языка

5. Встроенные механизмы валидации

* Типовая проверка через атрибут type
* Специфичные ограничения:
  + extensions для FilePath
  + options для Selection
  + Неявные диапазоны для Coordinate

6. Оптимизация для QML

* Прямое соответствие между JSON-описанием и QML-компонентами
* Использование свойств Qt-моделей для двустороннего биндинга
* Поддержка сигналов/слотов для реактивного интерфейса

7. Простота сериализации

* JSON как естественный формат для:
  + Сохранения/загрузки конфигураций
  + Передачи по сети
  + Интеграции с REST API

3.2. Основные классы в C++

3.2.1. Класс ISetting

Первым был реализован абстрактный класс в котором описаны свойства которыми должны обладать все настройки

#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*

Класс ISetting представляет собой фундаментальный интерфейс в вашей библиотеке, выполняющий несколько ключевых функций:

Абстракция настроек*.* Класс 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.

Использование SettingType в методе type() обеспечивает:

* Проверку типов на этапе компиляции
* Защиту от ошибок runtime через строгую типизацию
* Простую расширяемость для новых типов настроек

Методы get\_caption() и get\_description() следуют принципу явного определения отсутствия значения (возврат пустой строки вместо nullptr), что:

* Упрощает обработку в QML
* Избегает неопределенного поведения
* Соответствует принципам defensive programming

Аналогичный класс был создан для группы настроек ISettingsGroup

3.2.1. 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*

Краткое описание ключевых аспектов:

1. Назначение  
   Специализированный интерфейс для настроек с предопределенными вариантами выбора (например, выпадающие списки)
2. Основной функционал
   * get\_selection\_list() - возвращает доступные варианты в JSON-формате
   * is\_valid\_value() - проверяет соответствие значения допустимым вариантам

3.2.1. Фабричные классы ISettingParser ISettingsGroupParser ISettingsJsonParser

Представленные интерфейсы реализуют модифицированную версию паттерна "Фабричный метод" с элементами "Абстрактной фабрики". Рассмотрим ключевые аспекты:

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;

Композиция вместо наследования. 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

Двойной интерфейс парсинга. ISettingsJsonParser предлагает два варианта создания объектов:

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

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

Соблюдение SRP. Каждая фабрика отвечает только за один тип объектов:

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

Преимущества подхода

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

3.2.1. Model класс TabsModel

Система Model-View 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.

#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*

Класс TabsModel представляет собой центральный компонент системы управления настройками, выполняющий критически важную роль связующего звена между бизнес-логикой приложения и пользовательским интерфейсом.

TabsModel реализует концепцию иерархического представления настроек, организованных по принципу вкладок, где каждая вкладка соответствует логической группе параметров. Такой подход особенно востребован в сложных приложениях с многочисленными настройками, требующими четкой категоризации. Модель наследует от QAbstractListModel, что позволяет ей идеально 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

Такой подход позволяет легко создавать сложные интерфейсы настроек с использованием стандартных QML-компонентов, сохраняя при этом полный контроль над данными на C++ стороне.

ГЛАВА 4. ПЕРСПЕКТИВЫ РАЗВИТИЯ

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ