МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

Исследование возможности добавления палитры команд в произвольные приложения с использованием фреймворка Qt

Магистерская диссертация

Направление 01.04.02 Прикладная математика и информатика

Профиль Математическое и программное обеспечение вычислительных машин

Зав. кафедрой _____ д.т.н., проф. Г.В. Абрамов ___.___ 2020 г.

Обучающийся _____ Д.В. Польшаков

Руководитель _____ к.ф.-м.н., доц. М.К. Чернышов

Воронеж 2020

Список сокращений

БД — база данных

ПО — программное обеспечение

ГИП — графический интерфейс пользователя

API — application programming interface (программный интерфейс приложения)

Содержание

ВВЕДЕНИЕ

Из-за роста возможностей персональных компьютеров, программное обеспечение становится все сложнее и функциональнее. Такие группы программ как графические редакторы, браузеры, органайзеры и многое другое обрастают огромным числом функций. Для доступа к ним используются элементы ГИП.

Для быстрого доступа к функциям приложения, используются горячие клавиши. Они обычно создаются только для команд, которые используются чаще всего. Остальные же приходится выбирать вручную в интерфейсе.

Как бы хорошо ни был разработан интерфейс, число функций может оказаться настолько большим, что появляется проблема с поиском нужного элемента управления. Кроме того, некоторые системы поддерживают возможность добавления сторонних модулей. В таком случае место расположения элементов регулируется сторонними разработчиками, а не авторами оригинального приложения.

«Палитра команд» — технология, которую придумали для упрощения поиска элементов. Она представляет собой специальное окно в интерфейсе приложения, где отображаются все доступные функции. Иногда рядом с функцией отображается сочетание горячих клавиш, с помощью которых пользователь может её вызвать. Такой подход помогает пользователю легко запоминать новые сочетания, который он забыл или не знал раньше. В этом же окне есть поле для ввода поискового запроса.

Такая функциональность появилась в различных текстовых редакторах и средах разработки. Палитра команд оказалась достаточно удобной, поэтому позже энтузиасты разработали библиотеку с открытым исходным кодом Plotinus. Она позволяет добавлять такую функцию в приложения, которые используют фреймворк GTK. Данная библиотека опиралась на механизм, который предоставляет сами GTK, для расширения готовых приложений. Более подробный обзор приложений, которые используются палитру команд производится в разделе 1.1.

Целью данной работы было реализовать механизм, который позволял бы добавлять палитру команд в сторонние приложения без их пересборки. Из-за того, что такой механизм для GTK уже существует, был выбран другой (не меньший по распространенности) фреймворк — Qt.

Для достижения поставленной цели, нужно было исследовать возможность добавления функций в уже собранную программу, разработать библиотеку для расширения произвольных приложений и программу для координации работы.

В первой главе данной работы рассматриваются существующие приложения, которые используют палитру команд, проводится анализ способов добавления функций в существующее приложение, формулируются задачи для реализации и описывается архитектура взаимодействия всех элементов разрабатываемой системы.

Во второй главе рассматриваются детали и средства реализации каждого из элементов системы, протокол взаимодействия их друг с другом, и вспомогательные средства, которые были разработаны для решения технических задач.

В третьей главе производится анализ полученного решения: сравнение производительности приложения с использованием решения и без него, рассматриваются возможности дальнейших улучшений.

ГЛАВА 1. АНАЛИЗ ЗАДАЧИ

1.1. Обзор существующих реализаций палитр команд

Впервые палитра команд появилась 1 июля 2011 году в редакторе Sublime Text 2 [4]. Вслед за этим подобная функция была реализована в некоторых других программах. Таких, как:

- Atom[5],
- VSCode[6],
- JupyterLab[7].

Но это были лишь единичные случаи. В апреле 2017 года появилась альфа-версия приложения Plotinus[8], которое позволяет добавлять палитру команд в любое приложение, использующее графическую библиотеку GTK.

Таким образом можно наблюдать, что частные решения начинают заменяться более универсальными. Однако на текущий момент эти решения не позволяют покрыть большинство областей т.к. ограничены лишь программами с GTK, который используется не более чем в половине прикладных приложений для ОС Linux и занимает совсем малую долю среди приложений для ОС Windows.

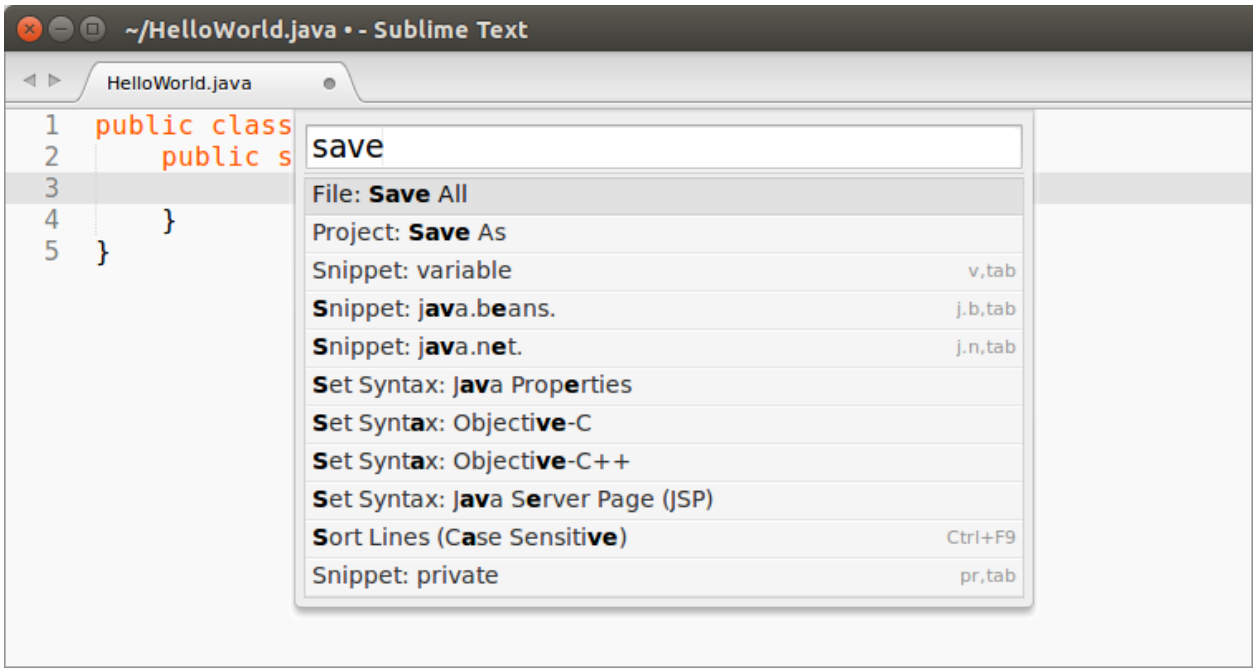

Рис. 1.1: Sublime Text

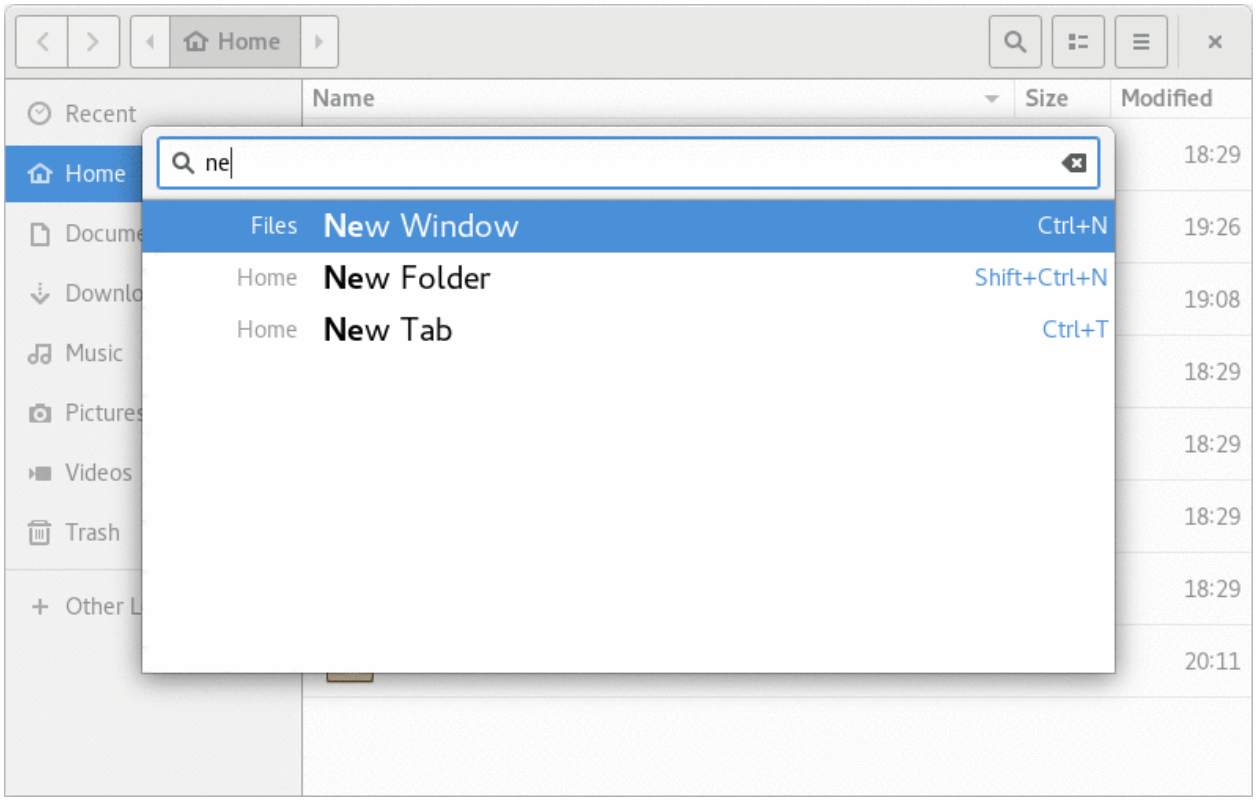

Рис. 1.2: Plotinus

1.2. Qt

Qt — это кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++[3]. Он содержит множество библиотек для упрощения реализации прикладных задач. Благодаря кроссплатформенности данный фреймворк позволяет запускать написанное с его помощью ПО на многих операционных системах путем обычной сборки проекта без внесения изменений в исходный код самой программы. Отличительной особенностью Qt является наличие метаобъектного компилятора, который запускается до основной компиляции и генерирует вспомогательный код. Такой подход позволил добавить частичную поддержку такого инструмента как рефлексия.

Qt включает в себя следующие модули:

- Qt Core — базовые классы, обеспечивающие рефлексию, работу со строками, механизмы владения и т.п. Используются всеми остальными модулями;
- Qt Network — классы, позволяющие легко писать переносимый код для работы с сетью;
- Qt SQL — классы, предоставляющие удобный программный интерфейс для работы с различными реляционными БД;
- Qt Multimedia — классы, для работы с данными (аудио и видео), устройствами (камеры, микрофоны);
- Qt GUI — классы, позволяющие реализовать приложения с графическим интерфейсом.

В ОС Linux библиотеки GTK и Qt являются двумя наиболее популярными средствами для реализации приложений с графическим интерфейсом. Т.к. средство для добавления палитры команд уже есть для GTK, в данной работе будет рассмотрена такая возможность для Qt.

События в Qt — это объекты, унаследованные от абстрактного класса `QEvent`. Они представляют действия, произошедшие внутри приложения, либо созданные в результате активности пользователя, о которых приложение должно знать. События могут быть получены и обработаны любым эк-

земпляром подкласса `QObject`, но они особенно актуальны для графических элементов.

Обычно события доставляются объектам через вызов виртуальных функций. Если разработчик хочет заменить функцию базового класса, он должен реализовать всю обработку самостоятельно. Однако, если нужно только расширить функциональность то разработчик должен реализовать нужное расширение, а затем вызывать базовый класс, чтобы обеспечить поведение по умолчанию для тех случаев, которые он не хочет обрабатывать.

Не всегда в классе имеется нужная функция для события. Наиболее распространенный пример — обработка нажатия клавиш, для которой нет соответствующей виртуальной функции. Для обработки таких событий нужно переопределить метод `QObject::event()`. Он является общим обработчиком событий, и позволяет выполнить дополнительные действия до или после обработки по-умолчанию.

1.3. Организация работы приложения

Система управления должна позволять контролировать множество приложений. Для ее реализации лучше всего подходит клиент-серверная архитектура. Для каждого целевого приложения запускается отдельный клиент, который занимается сбором информации об элементах управления и передает ее на сервер.

В качестве сервера будет выступать приложение, которое запускает целевые приложения вместе с клиентами. После этого сервер принимает входящее соединение от клиента и отображающее окно поиска элемента для текущего активного окна.

Для удобной работы окно поиска должно отображаться окно поверх текущего активного приложения. Палитра команд является инструментом для помощи пользователю в поиске нужной функции. Он может не знать точное наименование команды, поэтому в приложении должна быть поддержка нечеткого поиска.

1.3.1. Добавление логики в стороннее приложение

Разработчики любого приложения не могут учесть желания и капризы всех пользователей. Благодаря этому приложения не превращаются в комбайны, которые невозможно было бы поддерживать. Разработчики могут убрать какую-то деталь, которая нужна малому проценту людей. Ведь надо тратить время на исправление ошибок в ней. А иногда эти специфичные функции могут даже замедлять работу всего приложения. В таком случае пользователи могут захотеть добавить какую-то дополнительную логику или функцию в основное приложение.

Подходы делятся на два типа:

- добавление функции на этапе сборки приложения;
- добавление функции в момент выполнения программы.

Целью данной работы является добавление функциональности в максимальную группу приложений. Первый же подход исключает такую возможность для приложений с закрытым исходным кодом. К тому же при первым подходом может пользоваться только квалифицированный пользователь.

И то, ему пришлось бы пересобирать каждое приложение в которое он хотел бы добавить нужную функциональность.

Программный модуль подключаемый к уже существующему приложению называется плагином. Для добавления возможности подключения плагинов разработчики программы должны или написать свою систему плагинов или воспользоваться готовой. Так, например, библиотека GTK, начиная с третьей версии, предоставляет возможность запускать приложения с дополнительными модулями, которые могут расширять функциональность приложения. Этим воспользовались разработчики библиотеки Plotinus, реализовав возможность добавления палитры команд в любое приложение, использующее GTK.

Qt предоставляет возможность встраивать дополнительную функциональность в приложение, но для этого оно должно иметь специальный код по загрузке дополнительных модулей, который в большинстве случаев не используется (для подтверждения можно сравнить число github репозиториев использующих Qt[9] и использующих функцию Qt для работы с плагинами[10]. Соотношение примерно 1:100).

Кроме штатных средств добавления возможностей на уровне приложения есть и более низкоуровневые. Так например в Windows есть функция автоматизации интерфейса: UI Automation[11], которая изначально была добавлена для увеличения доступности приложений людям с ограниченными возможностями. С её помощью можно работать только с видимыми элементами интерфейса, но не получать внутренние состояния (что можно сделать через плагины), но зато доступен для любого приложения, использующего стандартные элементы управления. В случае использования Linux, к сожалению, нет такой возможности на уровне ОС или графической оболочки.

1.3.2. Внедрение модуля

Рассмотрим в общих чертах механизм работы графического приложения. На рис. 1.3 изображено как происходит взаимодействие между элементами графического приложения и пользователем.

Исходя из данной упрощенной схемы можно предложить еще одно способ добавить функциональность — создание события от графической библиотеки, которое будет передано приложению.

Рис. 1.3: Взаимодействие элементов ГИП и пользователя

1.3.3. Механизм подмены функций

При запуске приложения загрузчик получает из него список всех используемых динамических библиотек, загружает их в память. Затем получает адреса всех экспортированных функций динамической библиотеки и сохраняет их для последующего вызова.

Загрузчик ld, который используется в Linux и FreeBSD позволяет загружать дополнительные динамические библиотеки, кроме тех, кто запрашивает приложение. Эта дополнительная библиотека загружается раньше всех остальных, что позволяет ей подменять функции из других библиотек. Это происходит потому, что при поиске адреса определенной функции берется первый подходящий.

1.3.4. Способ получения информации об элементах

Для получения информации об элементах интерфейса можно загрузить специальную библиотеку, которая будет регистрировать создания, изменения и удаления элементов интерфейса. Затем собранная информация будет передавать на сервер для последующей работы.

Также данная библиотека может создавать ложные события по командам, приходящим с сервера. Таким образом можно имитировать нажатия кнопок, открытие меню и т.п.

Пользовательский интерфейс через специальное API сможет получать от сервера информацию о доступных элементах в текущем приложении. После того, как пользователь произвел выбор, вызывается специальная функция

на стороне сервера, которая приводит к отправке команды клиенту.

1.3.5. Реализация кода для подмены функции

Для внедрения библиотеки требуется реализовать заглушки функций Qt, которые будут вызывать специальный обработчик, а затем продолжать нормальное выполнение функции. Дополнительную сложность создает то, что библиотека Qt написана на языке C++, который из-за поддержки классов и перегрузок функций использует т.н. «искажение имен» (name mangling). Таким образом, чтобы создать обработчик, нужно сконструировать специальное имя функции исходя из названия класса, метода, набора параметров и возвращаемого значения.

Создание таких обработчиков является рутинной задачей в которой человек легко может допустить ошибку. Поэтому вместо ручного написания каждого обработчика нужно написать генератор, который может добавить нужные обработчики имея минимальный и необходимый набор данных (имя класса, метода и т.д.).

1.4. Постановка задачи

Требуется разработать набор программ, которые в комплексе будут решать следующие задачи:

- запускать целевые приложения в специальном окружении;
- собирать информацию о существующих элементах графического приложения;
- сохранять информацию о всех запущенных приложениях;
- отображать пользователю окно для поиска и выбора элемента;
- активировать выбранный пользователем элемент.

Исходя из приведенного выше анализа следует, что задачи должны быть сгруппированы в набор программ. Он должен быть реализован в виде следующих элементов:

1. Библиотека для внедрения и сбора информации в конкретном приложении.
2. Приложение для сохранения информации, полученной из нескольких приложения с библиотекой из п.1.
3. Графический интерфейс для запуска приложений и отображения окна палитры команд.

1.5. Архитектура системы

Подходящая архитектура для такой задачи была предложена мной в одной из прошлых работ[1]. Вся работа системы может быть рассмотрена в 3х частях:

1. регистрация изменений в элементах приложения;
2. посылка команды;
3. активация элемента.

Рассмотрим детально каждую часть.

1.5.1. Регистрация нового элемента

При запуске приложения, в него внедряется подгружаемый модуль, который переопределяет некоторые функции библиотеки Qt. Благодаря механизму работы загрузчика, целевое приложение будет на самом деле вызывать поддельные функции, вместо реальных.

Когда приложение создает элемент интерфейса или меняет описание уже существующего происходит вызов соответствующей функции, которую мы перехватываем. После этого наша библиотека посылает на сервер информацию о новом элементе или об изменении старого.

Когда все дополнительные действия сделаны, библиотека должны обеспечить стандартное поведение функции, которую она подменила. Для этого, используя механизмы загрузчика, она получает адрес настоящей функции и передает управление Qt. Графическая библиотека в свою очередь занимается формированием изображения и передает его на отрисовку в графическую подсистему X11.

На рис. 1.4 изображена диаграмма последовательности для этой процедуры.

1.5.2. Посылка команды

На рис. 1.5 изображена диаграмма последовательности для посылки команды. Рассмотрим её детально.

Пользователь, когда ему нужно, вызывает палитру команд и выбирает то, что его интересует. После этого приложение управления вызывает функ-

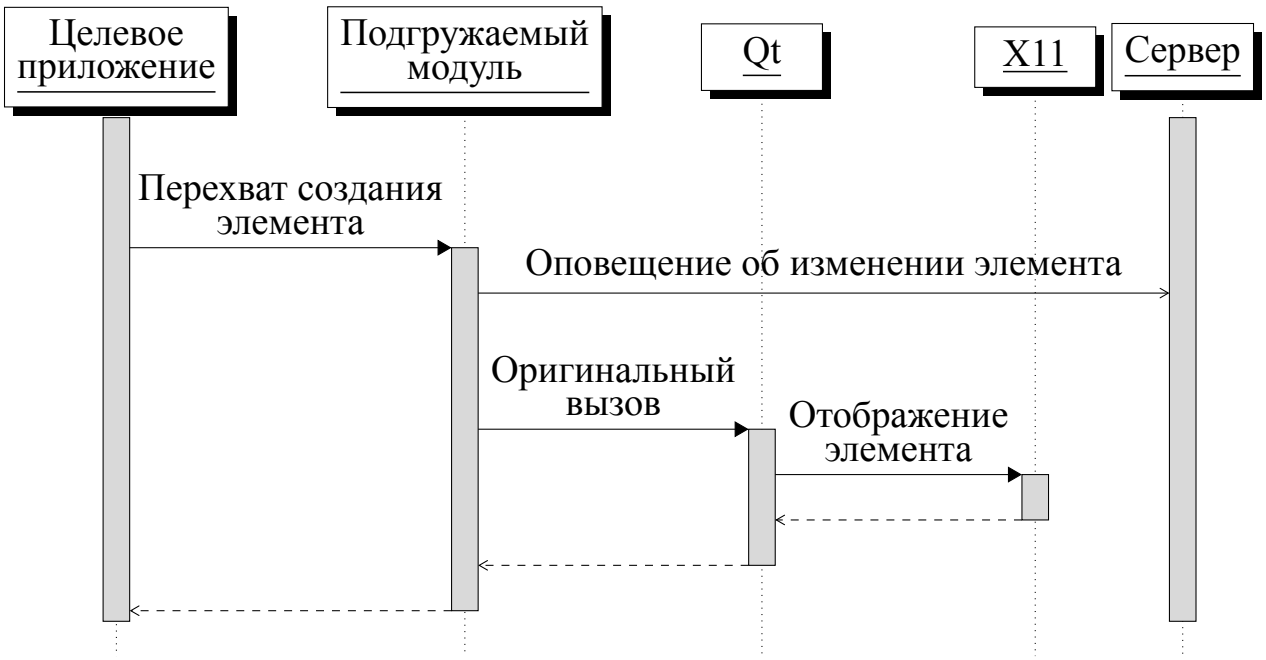

Рис. 1.4: Диаграмма последовательности регистрации изменений

цию на стороне сервера для активации команды. Сервер через сокет передает команду библиотеке, но она не может выступать инициатором действия, поэтому сервер посылает событие активации окна графической подсистеме. Это должно заставить систему послать событие приложению.

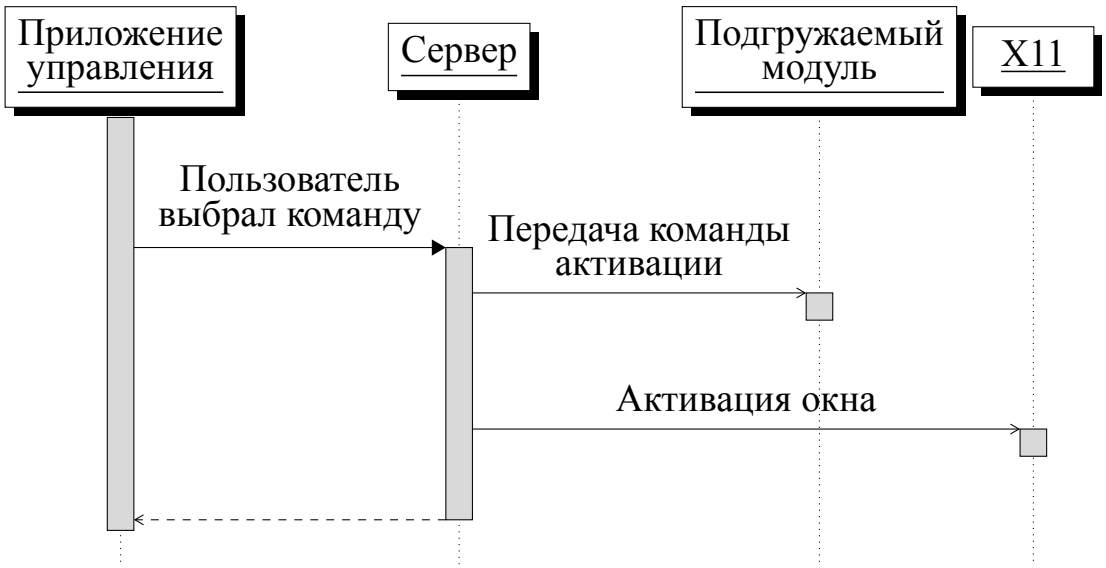

Рис. 1.5: Диаграмма последовательности посылки команды

1.5.3. Активация элемента

Когда X11 получает от сервера сообщение о том, что окно должно быть активировано, он информирует об этом графическую библиотеку. Пользова-

тельское приложение, которое использует Qt в качестве графической библиотеки, передает основное управление самому фреймворку, поэтому приемом сообщений занимается именно он.

Qt попытается передать приложению событие отрисовки. Его сможет перехватить подгруженный модуль и в этот момент выполнить дополнительные действия — активацию элемемента. Кроме активации элемента мы передаем событие в приложение, чтобы обеспечить стандартное поведение. Активация элемента производится штатными средствами Qt. Для каждого объекта способ активации свой.

На рис. 1.6 изображена диаграмма последовательности для процесса активации элемента.

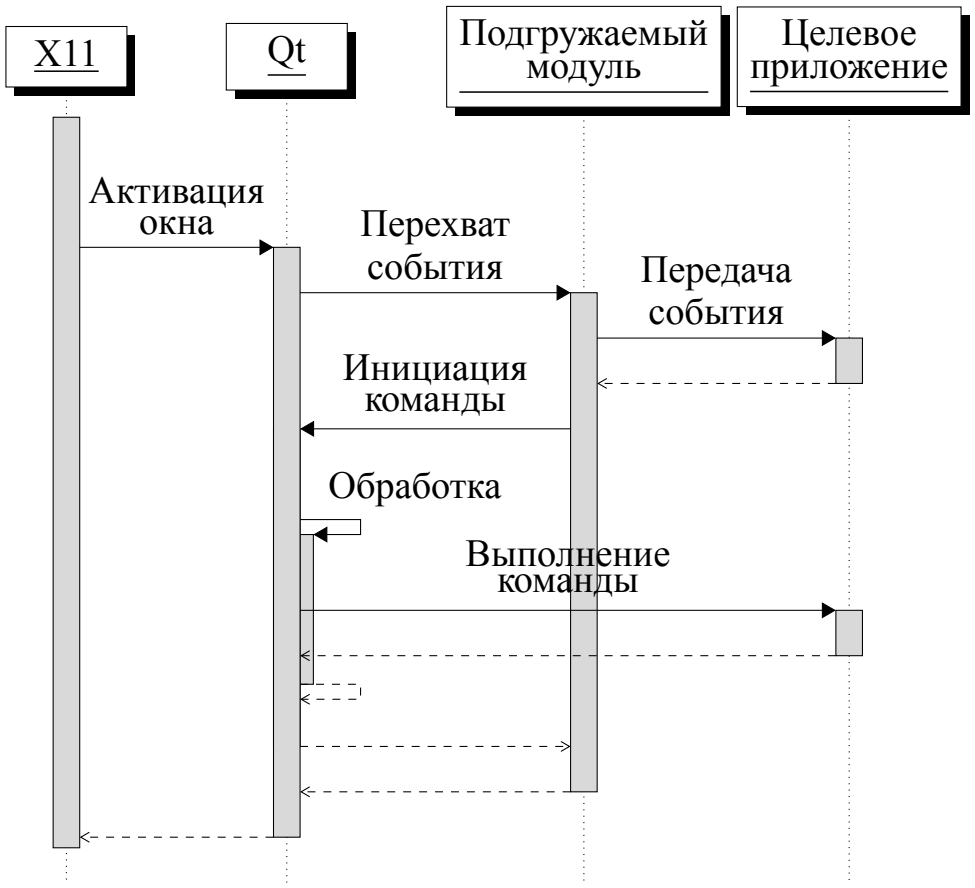

Рис. 1.6: Диаграмма последовательности активации элемента

1.5.4. Архитектура приложения управления

Программа управления должно выполнять две основные функции: запуск других приложений и отображение палитры команд. Поэтому с точки зрения архитектуры оно было разделено на соответствующие две части.

В каждой части было произведено разделение на часть логики и часть пользовательского взаимодействия. Такой подход позволяет менять отображение не затрагивая код логики и наоборот.

Затем все части соединяются в специальном интегрирующем модуле, который позволяет пользователю выбирать какой тип действия надо совершить.

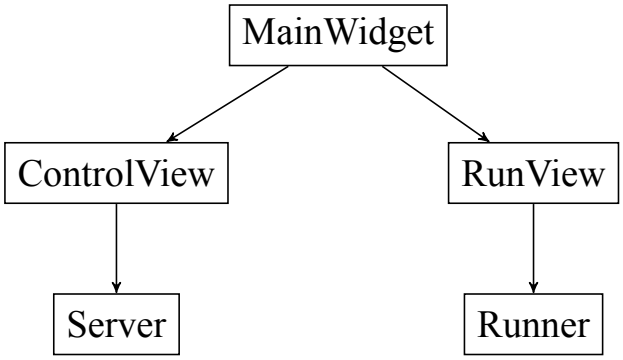

Рис. 1.7: Архитектура управляющего приложения

ГЛАВА 2. РЕАЛИЗАЦИЯ КОМПЛЕКСА ПРОГРАММ

2.1. Средства реализации

- Язык программирования C для написания внедряемой библиотеки;
- Язык программирования Python версии 3.8.3 для написания графического интерфейса и генератора кода;
- Система сборки CMake версии 3.2.
- Система управления версиями Git.

В данной работе используются два языка программирования по следующим причинам:

- Язык C предоставляет низкоуровневый интерфейс, который позволяет переопределить нужные функции во внедряемой библиотеке и в то же время без особой сложности в нем можно реализовать передачу данных в сокет.
- Язык Python, напротив, позволяет писать высокоуровневый код, что упрощает разработку прикладных приложений (в частности с графическим интерфейсом).

Во время разработки приложения управления на языке Python, с целью избежать дублирования кода, были использованы готовые внешние модули:

- **Qt** для реализации графического интерфейса;
- **xdk** для разбора файлов стандарта freedesktop;
- **system_hotkey** для назначения горячих клавиш на уровне операционной системы;
- **ewmh** для взаимодействия с графической подсистемой X11;
- **rofi** для отображения палитры команд и выполнения нечеткого поиска;
- **subprocess** для запуска внешних приложений.

Все модули доступны для установки с помощью Pip — официального пакетного менеджера Python для Linux.

2.2. Требования к программному и аппаратному обеспечению

Приложение предназначено для использования на IBM PC-совместимых компьютерах с операционной системой Linux.

Работы приложения требуются:

- аппаратное обеспечение согласно требованиям ОС;
- ОЗУ не менее 1 Гб;
- 1 Гб свободного места;
- наличие интерпретатора Python версии не ниже 3.0.

Требования к целевому приложению:

- приложение должно использовать графическую библиотеку Qt;
- должна быть произведена динамическая линковка с данной библиотекой.

2.3. Реализация генератора кода

На вход генератору подается файл, представляющий собой готовый к использованию исходный код, за исключением отсутствия функций-оберток. Для того, чтобы корректно их сгенерировать, в файле должна присутствовать информация, которая бы позволяла узнавать, какой обработчик для каждой функции должен быть применен. Для этого было принято решение добавить специальные комментарии, которые состоят из:

1. Опорного элемента (`//method`). Он требуется для того, чтобы отличать описательные комментарии от обычных.
2. Сигнатура метода в которой не указываются имена переменных.
3. Разделитель сигнатуры и названия функции-обработчика. В качестве разделителя выступает стрелка `->`.
4. Название функции-обработчика.

Пример комментария: `//method bool QWidget::event(QEvent*) -> checkEvent`

Вместо каждого такого комментария генератор подставляет код функции, реализующий следующий алгоритм:

Функция *методБиблиотеки(a, b, c)***:**
    если *реальнаяФункция == NULL* **тогда**
        реальнаяФункция = следующийСимвол("методБиблиотеки");
    конец
    если *инициализацияУспешна()* **тогда**
        функцияОбработчик(a, b, c);
    конец
    возвратить реальнаяФункция(a, b, c);
конец

Если бы этого было достаточно, то для подстановки можно было использовать макросы из языка Си. Однако сложность заключается в получении имени функции для подмены. Это связано с тем, что символы в исполняемом файле должны представлять собой просто идентификатор, который не предусматривает сам по себе типы параметров, возвращаемых значений,

имена классов и т.п. Стандарт языка C++ не говорит, как именно должно происходить данное преобразование, поэтому это будет зависеть от реализации. Существуют стандарты ABI (application binary interface, двоичный интерфейс приложения), которые определяют как должно происходить данное преобразование. Это позволяет библиотекам, собранным разными компиляторами с одним стандартом ABI, работать друг с другом. Для Linux одним из наиболее распространенных стандартов ABI является Itanium[12].

Т.к. в рамках данной работы не нужно генерировать имена с использованием пространств имен, виртуальных таблиц и т.п., была реализована только часть стандарта, описывающая получение имени функции и некоторых типов.

```
<имя-функции> ::= _Z <имя>
<имя> ::= <вложенное-имя> <параметры-функции>
<вложенное-имя> ::= N <префикс> <явное-имя> E
<префикс> ::= <префикс> <явное-имя>
          ::= <явное-имя>
<явное-имя> ::= <имя-из-исходного-кода>
            ::= <конструктор-или-деструктор>
<имя-из-исходного-кода> ::= <длина-идентификатора> <идентификатор>
<конструктор-или-деструктор> ::= C1
                             ::= D1

<параметры-функции> ::= <тип>
                    ::= <тип> <параметры-функции>
<тип> ::= <встроенный-тип>
      ::= <квалифицированный-тип>
      ::= <имя-класса-или-перечисления>

<встроенный-тип> ::= v # void
                 ::= i # int
                 ::= c # char
                 ::= b # bool

<квалифицированный-тип> ::= <квалификатор> <тип>
<квалификатор> ::= K # const
               ::= R # reference

<имя-класса-или-перечисления> ::= <имя>
```

2.4. Протокол связи между внедряемой библиотекой и сервером

Клиент и сервер общаются с помощью специального протокола. В рамках которого передаются команды.

```
<команда> ::= <имя-команды> <параметры-команды>
<имя-команды> ::= <строка>
<стока> ::= <длина-строки> <идентификатор>
<длина-строки> ::= uint32_t
<идентификатор> ::= "newApp"
                ::= "setWidgetText"
                ::= "remove"
                ::= "activated"
                ::= "setWidgetWindow"
                ::= "activate"
```

Рассмотрим существующие команды:

2.4.1. Регистрация приложения

Идентификатор: «newApp».

Параметры:

1. идентификатор процесса (8 байт);
2. адрес сокета для общения (строка).

Данная команда посылается от клиента к серверу в самом начале. Она нужна затем, чтобы сервер дифференцировал различные приложения при отправке команд в дальнейшем.

2.4.2. Установка текста элемента

Идентификатор: «setWidgetText».

Параметры:

1. идентификатор процесса (8 байт);
2. идентификатор элемента (8 байт);
3. текст элемента (строка).

С помощью данной команды клиент сообщает серверу о добавлении или изменении текста элемента. Если идентификатор уже использовался, то считается, что текст виджета изменился, иначе, что элемент только добавлен.

2.4.3. Удаление элемента

Идентификатор: «remove».

Параметры:

1. идентификатор процесса (8 байт);
2. идентификатор элемента (8 байт).

Используется, когда клиент считает, что элемент больше не доступен для активации. После выполнения данной команды, сервер должен перестать выдавать элемент в списке с запросом всех доступных виджетов. Идентификатор считается свободным и может быть повторно использован для новых элементов.

2.4.4. Сообщение об активации окна

Идентификатор: «activated».

Параметры:

1. идентификатор процесса (8 байт);
2. идентификатор окна (4 байта).

Клиент посылает данную команду, когда сменилось активное окно в приложении. Это требуется для того, чтобы сервер мог позже вернуться к последнему активному окну. Идентификатор окна представляет собой атрибут *windowId* графической подсистемы X11. Сервер должен использовать его для последующей работы с окном.

2.4.5. Привязка элемента к окну

Идентификатор: «setWidgetWindow».

Параметры:

1. идентификатор процесса (8 байт);
2. идентификатор элемента (8 байт);
3. идентификатор окна (4 байта).

Приложения могут создавать элементы, которые не привязаны к конкретному окну, а привязывать их позже. Или перемещать существующие элементы между окнами в целях оптимизации. Клиент должен посылать данную команду, чтобы сообщить серверу, что определенный элемент относится к конкретному окну. Серверу это нужно, чтобы понимать, какие элементы доступны в текущем окне.

2.4.6. Активация элемента

Идентификатор: «activate».

Параметры:

1. идентификатор элемента (8 байт);

Эта команда посылается от сервера клиенту, чтобы активировать элемент (нажать кнопку, открыть меню и т.п.)

2.4.7. Обработка ошибок

В случае, если в любой команде, использующей идентификатор элемента, приходит значение, которое не было зарегистрировано или было удалено, сервер должен проигнорировать команду.

2.5. Реализация библиотеки для внедрения

Библиотека, которая будет использоваться для сбора информации, должна реализовывать часть функций аналогично библиотеке Qt и выполнять следующие функции:

- выполнять регистрацию приложения;
- фиксировать создание объектов (в Qt элементы ГИП называются виджетами);
- фиксировать смену описания объекта;
- фиксировать перемещение элементов между окнами;
- фиксировать удаление объектов;
- активировать элемент по команде.

2.5.1. Регистрация приложения

Регистрация приложения должна происходить до создания первого объекта, но универсального метода, который вызывался бы в самом начале и был бы всегда доступен для переопределения, к сожалению, нет. Поэтому каждая функция-обертка в начале проверяет, была ли проведена инициализация. И если нет, то проводит ее, создавая сокет для получения команд от сервера и посылая ему команду регистрации текущего приложения.

2.5.2. Создание объекта

В рамках данной работы была реализована система регистрация создания флажков (`QCheckBox`), кнопок (`QPushButton`) и «действий» (`QAction`). «Действия» в Qt это абстракция именованной команды, которая может быть использована в ГИП[13]. Например, они применяются в главном меню, в контекстном меню, в качестве обработчиков горячих клавиш и т.п.

Создание объектов всегда происходит через вызов конструкторов классов. Не все конструкторы позволяют указывать текст описания элемента. Поэтому для реализации более общего случая конструкторы с указанием текста рассматриваются как выполнение двух действий: создание объекта и установка текста.

2.5.3. Смена описания

Почти все виджеты в Qt позволяют менять связанный с ними текст (надпись на кнопке, название пункта меню и т.п.) во время исполнения. Для всех указанных выше типов был переопределен метод `setText`. При его вызове, клиент посылает на сервер команду установки текста элемента.

2.5.4. Перемещение элементов между окнами

Qt не имеет доступного для переопределения метода, который позволял бы отслеживать перемещения объекта. Поэтому в библиотеке пришлось переопределить метод `QWidget::event`. Он вызывается при всех событиях, на которые должен отреагировать хотя бы один виджет. А фиксация изменения окна у элемента, происходит в два этапа:

1. Получить список всех окон.
2. Для каждого окна проверить все его дочерние элементы.

Qt использует отличные от X11 понятия «окна». Так, например, элемент может быть отображен без явного создания соответствующего окна. Тогда на уровне Qt будет просто виджет, а на уровне X11 --- окно. Поэтому отдельной задачей является получение списка всех окон в терминах X11. Таковыми являются все виджеты верхнего уровня, у которых нет родителя.

Затем запускается рекурсивный алгоритм, который проверяет, есть ли среди дочерних элементов новые.

```
Функция привязатьЭлементКОкну(элемент, новоеОкно):
    староеОкно = окноЭлемента[элемент];
    если староеОкно == NULL или староеОкно != новоеОкно тогда
        послатьКомандуСменыОкна(элемент, новоеОкно);
        окноЭлемента[элемент] = новоеОкно;
    конец
    для каждого дочернийЭлемент в элемент.подэлементы()
     выполнять
        привязатьЭлементКОкну(дочернийЭлемент, новоеОкно);
    конец
конец
```

Такой метод не слишком оптимален. В рамках улучшения данного решения стоит поискать возможность усовершенствовать его.

2.5.5. Удаление элементов

В текущей реализации элемент считается недоступным после его удаления из памяти (т.е. при вызове деструктора), но для удобства работы правильнее было бы обрабатывать события скрытия и отображения элемента.

2.5.6. Активация по команде

Как было указано в моей прошлой работе[1], библиотека не может выступать инициатором действия, она в состоянии только реагировать на события. Поэтому для активации элемента требуется событие, которое, например, посылает сервер. В текущей реализации была выбрана функция обработки событий как наиболее часто вызываемая. Именно в ней проверяется наличие команды активации.

2.5.7. Устройство библиотеки

Библиотека состоит из функций. Они делятся на два типа: обработчики и вспомогательные. Для того, чтобы сохранять состояния между вызовами функций приходится использовать глобальные переменные.

Данные в библиотеке

- `g_server` — сокет для отправки команд на сервер;
- `g_client` — сокет для получения команд от сервер;
- `g_handlers` — ассоциативный массив, ставящий каждому объекту интерфейса в соответствие функцию активации. Это требуется для того, чтобы на этапе получения команды «activate» от сервера, библиотеке не приходилось проверять все возможные типы объекта;
- `g_widgetWindow` — ассоциативный массив, ставящий каждому объекту интерфейса в соответствие окно, к которому объект принадлежит.

2.5.8. Функции обработчики

- `checkEvent` — функция обработки событий. Заменяет собой `QWidget::event`. Посылает серверу команду о том, что окно активировано, когда в качестве параметра приходит `QEvent::WindowActivate`;

- `registerButton`, `registerCheckbox` — функция обработки создания кнопки и флажка. Заменяет собой конструкторы классов `QPushButton` и `QCheckBox`. Посылает серверу команду о создании нового виджета.

- `registerAction`, `registerActionWithIcon` — функция обработки создания кнопки. Заменяет собой два перегруженных конструктора класса `QAction`. Посылает серверу команду о создании нового события.

- `updateButtonText`, `updateActionText` — функция обработки изменения текста. Заменяют собой функции `setText` классов `QPushButton` и `QAction`.

Вспомогательные функции

- `updateWidgetsWindowsRecursive` — функция рекурсивного обхода объектов (на основе механизма владения, предоставленном Qt).

- `updateWidgetWindow` — функция для проверки изменения и первой установки родительского окна у элемента.

- `doUpdateWidgetWindow` — функция для отправки серверу информациио привязке виджета к окну.

- `activateWidget` — функция обработки команды активации элемента. Проверяет наличие данных в сокете, разбирает команду и вызывает функцию активации основываясь на данных из `g_handlers`.

- `setWidgetText` — функция для отправки серверу команды для информирования об изменении текст виджета.

- `initInject` — функция инициализации библиотеки. Подключается к сокету сервера, создает свой сокет через который будет получать коман-

ды, посылает серверу команду, в которой сообщает свой идентификатор процесса и адрес своего сокета.

- `getXProperty` — функция для получения свойства окна из графической системы X11. В качестве параметра принимает дескриптор экрана, идентификатор окна и имя свойства. Возвращает указатель на массив данных, представляющих собой значение свойства.

- `getLongProperty` — функция для получения значения свойства, которое представимо 4я байтами.

- `getWindowId` — функция получения идентификатора текущего активного окна. Предполагается, что будет использоваться при получении события активации окна, поэтому будет возвращать идентификатор текущего окна.

- `getPid` — функция получения идентификатора процесса прикладного приложения в который было произведено внедрение.

- `sendData` — функция для отправки «сырых» данных.

- `sendString` — функция для отправки строк. В соответствии с протоколом, сначала передается длина строки, потом строка в кодировке ASCII.

2.6. Реализация сервера

Для реализации функциональности сервера был написан одноименный класс. Вся основная работа данного класса происходит в отдельном потоке, которым он сам и управляет. При инициализации класс создает файловый сокет и поток, для последующего исполнения.

Непосредственно для своей работы класс `Server`, использует вспомогательные классы:

- `WidgetInfo` — описание одного элемента графического интерфейса. Состоит из:
  - `addr` — идентификатор, который указан в протоколе. На практике используется адрес объекта внутри приложения;
  - `text` — текст, соответствующий элементу (надпись на кнопке, название пункта меню).
- `Window` — описание одного окна приложения. Состоит из
  - `wid` — идентификатор окна, используемый для работы с X11;
  - `widgets` — список всех элементов в этом окне.
- `App` — описание приложения. Состоит из
  - `pid` — идентификатор, который указан в протоколе; На практике используется системный идентификатор процесса;
  - `client` — сокет, в который пишет сервер, для связи с приложением;
  - `windows` — список окон приложения;

  Класс `Server` имеет следующие поля:

- `applications` — список всех подключенных приложений (см. `App` выше);
- `last_window_id` — идентификатор последнего активного окна. Используется для возврата фокуса после отображения палитры команд;
- `socket` — сокет сервера, в который пишут клиенты;

- `running` — флаг, сигнализирующий о том, должен ли сервер прекращать работу;
- `thread` — объект для управления потоком.

Все вышеперечисленные поля являются доступными только для данного класса. Поэтому в коде их имена начинаются с двух символов подчеркивания.

Класс `Server` имеет следующие методы:

- `start` — функция запуска потока;
- `stop` — функция остановки выполнения основного обработчика потока. Выставляет флаг `running` в значение `false`, чтобы следующая итерация не запустилась;
- `loop` — основной цикл потока. Ожидает данные из сокета и при их получении вызывает функции для обработки команд;
- `handle_cmd` — функция обработки команд от клиентов. Занимается разбором данных, пришедших из сокета. После чего выбирает функцию обработчика конкретной команды;
- `add_new_app` — функция обработчик команды «добавить приложение». Из полученных данных создает объект `App` и добавляет его в информационную базу;
- `set_widget_text` — функция обработчик команды «установить текст элемента». Проверяет наличие элемента по идентификатору. Если элемент не найден, создает новый, иначе обновляет уже существующий;
- `set_widget_window` — функция обработчик команды «привязка элемента к окну». Удаляет из старого окна, добавляет в новое. Если нового нет — создает;
- `remove` — функция-обработчик команды «удалить элемент». Находит окно с элементом и удаляет из него информацию о нем.

- `activated` — функция обработчик сообщения «об активации окна». Сохраняет пришедший идентификатор окна в поле `last_window_id` для дальнейшего использования;
- `activate` — функция, выполняющая поиск элемента по его названию и вызывающая функцию активации элемента;
- `activate_widget` — функция активации элемента. Посылает команду клиенту и активирует окно через X11 для передачи события;
- `get_options` — функция получения всех доступных вариантов для активации (список строк);
- `get_app` — вспомогательная функция для получения приложения по идентификатору процесса;
- `find_window_by_wid` — вспомогательная функция для получения окна по его идентификатору.

Также есть ряд функций, которые не входят в класс, но используются им:

- `recv_uint32`, `recv_uint64` — функции чтения из сокета 4-х и 8-и байт соответственно;
- `recv_text` — функция получения строки, описываемой протоколом;
- `activate_widget` — функция, посылающая в сокет клиента команду, на активацию элемента;
- `activate_window` — функция, посылающая графической подсистеме X11 команду на активацию окна.

2.7. Реализация приложения управления

Основные функции приложения управления, это

- предоставление интерфейса для запуска приложений в специальном окружении, которое позволяет использовать подмену библиотеки;
- отображение палитры команд, в которой пользователь может выбрать действие для выполнения.

Из-за того, что приложение должно выполнять всю свою работу и взаимодействовать с другими приложениями в фоне, оно не требует основного окна. Вместо этого отображается иконка в области уведомлений. Контекстно меню состоит из пунктов:

1. Вызов палитры команд.
2. Запуск приложений.
3. Выход.

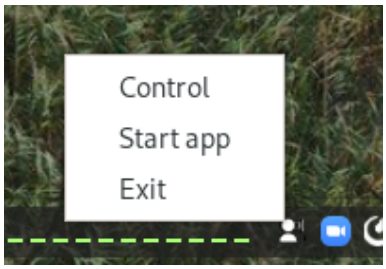

Рис. 2.1: Приложение в области уведомлений

Использование контекстного меню для данной задачи не удобно, в приложение были введены сочетания горячих клавиш **Ctrl + Shift + D** и **Ctrl + Shift + S** для запуска приложений и отображения палитры команд соответственно.

Из-за того, что пользователь не всегда помнит точное наименование пункта меню или названия команды, ему будет сильно удобнее использовать нечеткий поиск (fuzzy search). Для его выполнения был использован модуль python rofi, который позволяет отобразить всплывающее окно со списком элементов и выполнить в нем нечеткий поиск.

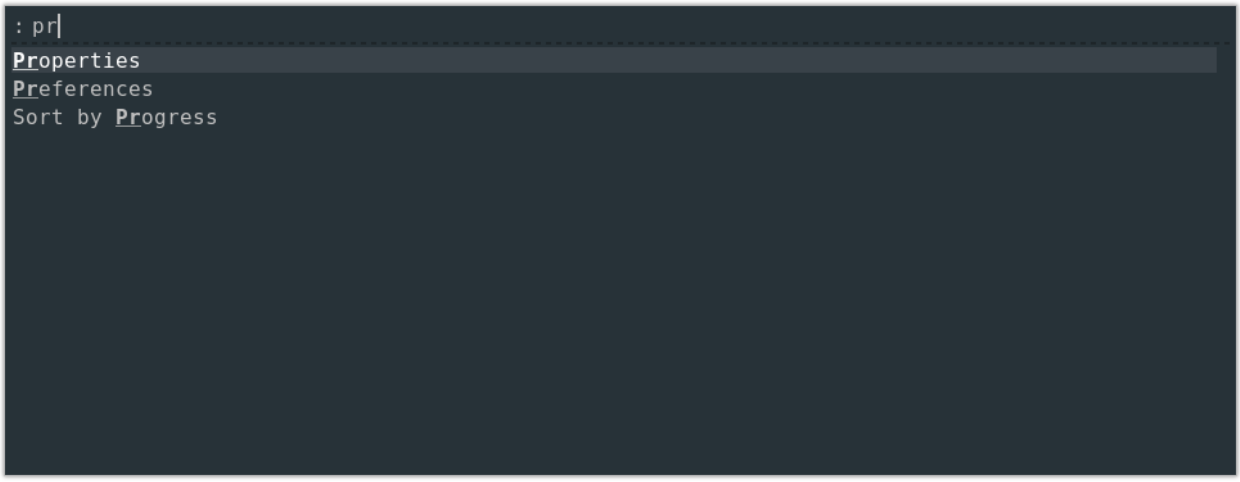

Рис. 2.2: Палитра команд

Чтобы упростить пользователю запуск программ, приложение при возможности должно само получить список доступных программ, а не заставлять пользователя самому создавать список таких программ. В большинстве дистрибутивов ОС Linux используется стандарт freedesktop, позволяющий в т.ч. описывать пользовательские приложения, которые можно запустить из меню приложений. Поэтом приложение управления сканирует директорию `/usr/share/applications`, считывает оттуда все файлы с расширением `.desktop` и на основе полученной информации составляет справочник доступных приложений. Затем с помощью того же модуля rofi, отображает меню выбора приложения для запуска.

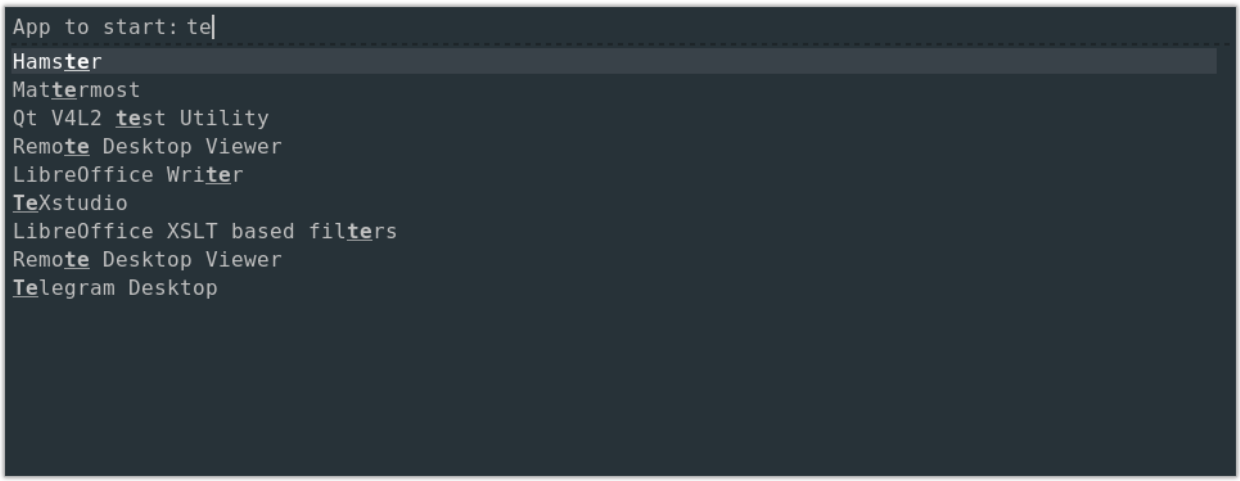

Рис. 2.3: Запуск приложения

ГЛАВА 3. АНАЛИЗ РЕЗУЛЬТАТОВ

3.1. Анализ производительности

3.1.1. Синтетическое тестирование

Синтетическим тестированием называют запуск стандартизированных тестов, на которых производятся замеры скорости работы тестируемой программы с целью получения объективных критериев показывающих производительность системы.

В данной работе было разработано средство для добавления дополнительной функциональности в существующее приложение. Из этого следует, что мы можем поменять уже существовавшее поведение приложения. Из-за того, что новая библиотека сканирует все объекты для определения их иерархической структуры, оно добавляет функцию сложности $O(N)$ для действий интерфейса, а из этого следует, что при увеличении числа графических элементов, производительность будет деградировать.

Для произведения замеров было разработанно тестовое приложение. Оно принимает в качестве параметра число окон, которые должно открыть. Каждое окно состоит прямоугольника с $N \times M$ кнопками, где значения N и M заданы на этапе сборки. Это приложение при запуске замеряет среднее время, которое требуется для открытия одного окна.

На рис. 3.1 можно видеть, что в нормальном режиме работы время открытия окна примерно постоянно и составляет 5.6 ± 1 секунду.

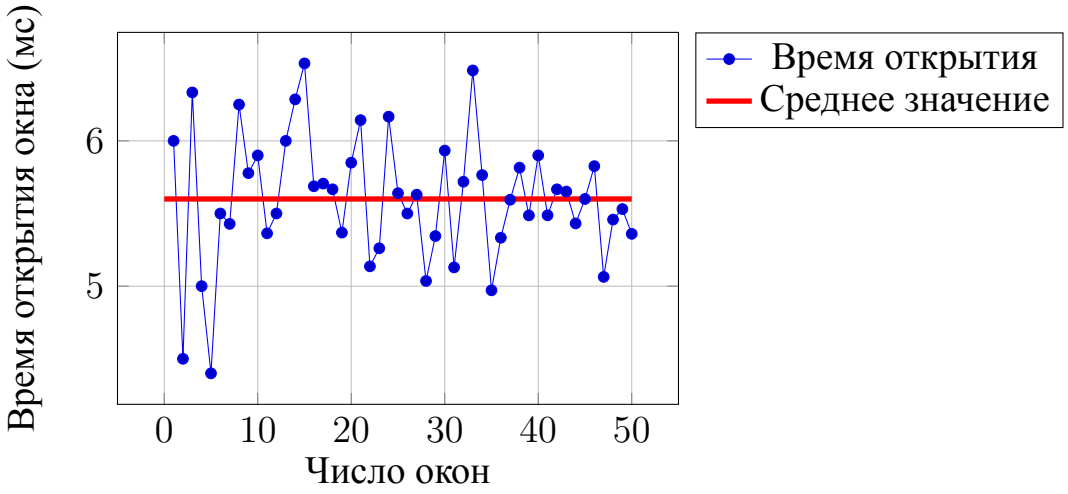

Рис. 3.1: Скорость открытия окна без библиотеки

На рис. 3.2 можно видеть, что при запуске приложения с использованием библиотеки для сбора информации, среднее время открытия одного окна

начинает линейно расти при увеличении числа элементов (окон и кнопок в этих окнах).

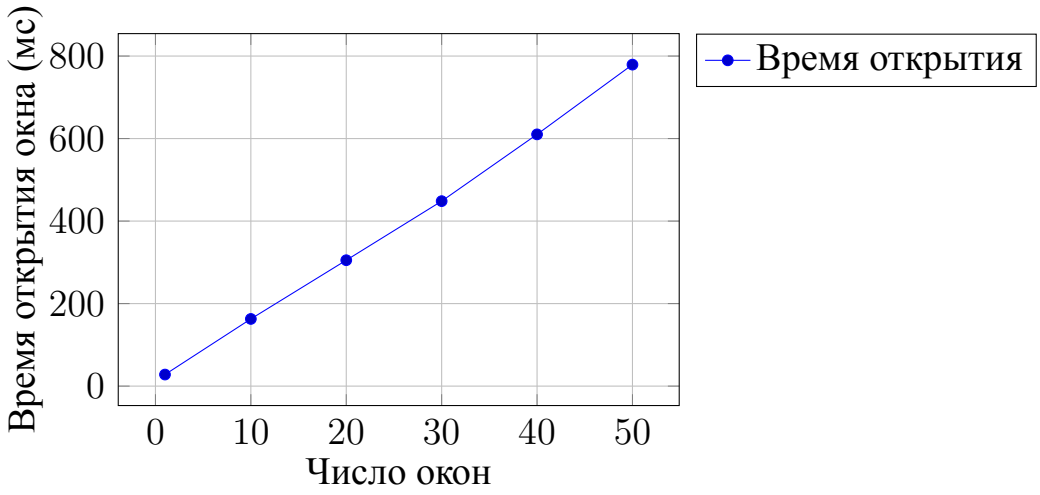

Рис. 3.2: Скорость открытия окна с библиотекой

3.1.2. Ручное тестирование

Кроме синтетического производилось и ручное тестирование в прикладных приложениях. Для этого через программу управления запускалось прикладное приложение и в нем производились активация хотя бы одной кнопки и одного из элементов главного меню. Во время такой проверки падение производительности не ощущалось.

3.2. Возможные усовершенствования

После реализации базового функционала осталось место для для дальнейших исследований и улучшений как в плане расширения функционала так и улучшения существующего.

В рамках выполненной работы была добавлена регистрация простых и наиболее распространенных элементов ГИП (кнопки, флажки, пункты меню). Однако для удобства пользователя стоит расширить реализацию и добавить регистрацию более сложных элементов (вкладки, таблицы), а также добавить поддержку универсальных команд для работы с окнами: закрыть, минимизировать, свернуть.

Ранее я рассматривал [2] возможность получения дополнительной информации о графических элементах управления. При использовании Qt разработчик может указывать описание к элементу, которое предназначено для людей с ограниченными возможностями. Эти данные могут быть использованы при поиске команды в палитре.

Также стоит рассмотреть возможность расширения числа перехватываемых функций, чтобы избавится от использования алгоритмический не оптимальной функции для задачи установления иерархии.

Как указывалось ранее, в палитре команд иногда указывают сочетание горячих клавиш, если оно привязано к команде. Qt предоставляет штатное средство для регистрации горячих клавиш — класс `QShortcut`. В случае, когда разработчик корректно настроил горячие сочетания для действий `QAction`, библиотека в состоянии установить связь между ними.

Система сигналов и слотов — это механизм коммуникации между объектами интерфейса, используемый в Qt. Сигнал срабатывает в момент определенного события (нажатия кнопки, клик мыши), а слот – это функция, которая вызывается в ответ на определенный сигнал. Преимуществом такого подхода является типобезопасность и слабосвязанность: класс, вырабатывающий сигнал ничего не знает о том, какие слоты его получат.

В библиотеку может быть добавлена логика для анализа связей сигналов и слотов, чтобы находить дополнительные связи: если в качестве источника сигнала выступает `QShortcut`, а в качестве приемника — известный элемент интерфейса, то можно связать грячие клавиши с активацией элемента.

В ОС Linux последнее время начинает набирать популярность графическая система Wayland как замена X11[14]. Qt уже адаптирован для работы с ним. Однако решение, представленное в данной работе опирается на механизмы X11 для инициации действия внутри стороннего приложения. Стоит рассмотреть возможность использования более универсального механизма — POSIX сигналы. В системах совместимых с POSIX, существуют специальные сигналы, которые пользователь может переопределить для своих целей. Они используются достаточно редко графическими приложениями, поэтому библиотека может попытаться зарегистрировать свой обработчик сигнала, а затем сервер будет посылать этот сигнал.

ЗАКЛЮЧЕНИЕ

В данной работе была поставлена задача разработать комплекс программ, который позволил бы пользоваться палитрой команд в произвольных приложениях, разработанных с использованием графической библиотеке Qt.

В ходе работы были подробно рассмотрены способы решения проблемы с получением информации из существующего приложения

Был реализован комплекс приложений, который позволяет отображать палитру команд в произвольных Qt приложениях, опираясь на информацию, которую разработчик оригинального приложения связал с элементом интерфейса.

Как говорилось ранее, палитра команд является удобной частью интерфейса и начинает появляться все в большем числе приложений. Полученные в ходе данной работы результаты могут найти практическое применение в системах основанных на Linux.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Использование перехвата вызова функций графической библиотеки для управления приложением с графическим интерфейсом пользователя --- Д. В. Польшаков

[2] Голосовое управление сложными системами --- Д. В. Польшаков

[3] Qt Framework - One framework to rule all! URL: https://www.qt.io/product/framework (дата обращения: 24.05.2020).

[4] Sublime Text 2 - Sublime Text --- URL: https://www.sublimetext.com/2 (дата обращения: 24.05.2020).

[5] command-palette --- URL: https://atom.io/packages/command-palette (дата обращения: 24.05.2020).

[6] Visual Studio Code User Interface --- URL: https://code.visualstudio.com/docs/getstarted/userinterface#_command-palette (дата обращения: 24.05.2020).

[7] Command Palette - JupyterLab 2.1.3 documentation --- URL: https://jupyterlab.readthedocs.io/en/stable/user/commands.html (дата обращения: 24.05.2020).

[8] Releases - plotinus/releases --- URL: https://github.com/p-e-w/plotinus/releases (дата обращения: 24.05.2020).

[9] Search - ``#include <QObject>" --- URL: https://github.com/search?l=C%2B%2B&q=%22%23include+%3CQObject%3E%2 (дата обращения: 24.05.2020).

[10] Search - ``#include <QPluginLoader>" --- URL: https://github.com/search?l=&q=%22%23include+%3CQPluginLoader%3E%22+la (дата обращения: 24.05.2020).

[11] UI Automation Overview - Win32 apps | Microsoft Docs --- URL: https://docs.microsoft.com/en-us/windows/win32/winauto/uiauto-uiautomationoverview?redirectedfrom=MSDN (дата обращения: 24.05.2020).

[12] Itanium C++ ABI --- URL: https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling (дата обращения: 24.05.2020).

[13] QAction Class | Qt Widgets 5.15.0 --- URL: https://doc.qt.io/qt-5/qaction.html (дата обращения: 24.05.2020).

[14] The Linux graphics stack from X to Wayland | Ars Technica — URL: https://arstechnica.com/information-technology/2011/03/the-linux-graphics-stack-from-x-to-wayland/ (дата обращения: 24.05.2020).

[15] Компьютерные сети. 5-е изд. — СПб.: Питер, 2012. — 960 с.: ил.

[16] Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.: ил