СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc501673251)

[1 АНАЛИЗ ИСХОДНЫХ ДАННЫХ 6](#_Toc501673252)

[2 ПРОГРАММНОЕ ПРОЕКТИРОВАНИЕ 7](#_Toc501673253)

[2.1 Проектирование «обёртки» 7](#_Toc501673254)

[2.2 Проектирование тестового приложения 17](#_Toc501673255)

[3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ 20](#_Toc501673256)

[3.1 Детальная реализация функциональных частей ПО 20](#_Toc501673257)

[3.1.1 Детальная реализация «обёртки» 20](#_Toc501673258)

[3.1.2 Детальная реализация тестового приложения 38](#_Toc501673259)

[3.2 Сопроводительная документация 42](#_Toc501673260)

[3.3 Анализ ПО 42](#_Toc501673261)

[3.4 Тестирование ПО 43](#_Toc501673262)

[ЗАКЛЮЧЕНИЕ 48](#_Toc501673263)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 49](#_Toc501673264)

[ПРИЛОЖЕНИЕ А 50](#_Toc501673265)

[ПРИЛОЖЕНИЕ Б 54](#_Toc501673266)

# ВВЕДЕНИЕ

Написание программного обеспечения для разных ОС – сложная задача. Главное препятствие, стоящем перед программистом: у каждой ОС есть свой API, которое не похоже на API других ОС, что препятствует переносимости кода. Однако каждая ОС по-своему раскрывает возможности оборудования и по-своему развязывает руки программистам, что позволяет им писать по-настоящему эффективный код, реализующий необходимый функционал.

Современное программирование стало гораздо проще благодаря появлению новых, более высокоуровневых языков программирования, позволяющих создавать более высокоуровневые абстракции из более примитивных типов. Особую роль сыграло появление высокоуровневых языков программирования, выполняемых в виртуальных средах исполнения, таких как Java [1] и C# [2]. Эти языки программирования позволили программисту перестать думать о особенностях разных ОС, а, вместо этого, сосредоточить своё внимание на написании эффективного и чистого кода. Однако эти языки программирования имеют и свои недостатки – при выполнении программы их промежуточный код необходимо компилировать в исполняемый код, что немного замедляет работу приложения. Поэтому в настоящие время до сих пор актуальны такие языки программирования, как C [3] и C++ [4]. Так же эти языки используются для написания кода, специфичного для конкретной ОС, в кроссплатформенных языках программирования.

Моей задачей является написание объектно-ориентированной обёртки на Win32 API для создания высокоуровневой абстракции над API windows, которая к тому же будет легко интегрироваться в код, написанный на C++ с вызовами Win32 API процедур.

Объектно-ориентированный подход к программированию популярен на сегодняшний день. Он облегчает написание кода за счёт введения абстракций, которых так не хватало процедурно-ориентированному подходу. Однако существует не так уж и много C++ библиотек, которые могли бы предоставить полное покрытие Win32 API, предоставить интерфейс для создания GUI и при этом не превратиться в фреймворк. Есть широко известный фреймворк, который был разработан самой компанией Microsoft - Microsoft Foundation Classes (далее MFC) [5]. Этот фреймворк развивается достаточно давно и успел обрасти как полезным функционалом, так и лишними запутанными связями, что препятствует продолжению активного развития фреймворка без опасения «что-то сломать». К тому же сам факт того, что MFC – фреймворк, заставляет хорошенько подумать перед тем как выбрать его. Это связано с тем, что фреймворки сковывают программиста и лишают его возможности «играть по своим правилам».

Так же существует ещё одно популярное решение – Windows Template Library (далее WTL) [6]. WTL – это свободно распространяемая библиотека шаблонов для написания GUI приложений для операционной системы Windows, однако данная библиотека решает вопрос построения GUI и никак не решает всего остального.

Популярный фреймворк QT [7] является лучшим решением для написания прикладных программ с GUI, однако в нём нет возможности прямого использования API операционной системы из-за своей кроссплатформенности.

Одним из самых популярных решений является библиотека GIMP ToolKit plus (далее GTK+) [8]. GTK+ предоставляет простой интерфейс для разработки графических приложений, однако данная библиотека написана на языке программирования C, что заметно в структуре программ, использующих эту библиотеку.

wxWidgets – кроссплатформенная библиотека для разработки кроссплатформенных приложений [9]. Библиотека не только отлично справляется со своей основной задачей – эффективное построение UI, но и ещё позволяет выполнять ряд других задач. wxWidgets позволяет сосредоточить всё внимание на написание логики приложения, а не на построение его базового интерфейса. Так же эта библиотека распространяется под лицензией LGPL, что позволяет внедрять решения, основанные на wxWidgets в закрытые проекты. Однако вновь появляются ограничения, связанные с кроссплатформенностью данной библиотеки.

Все рассмотренные библиотеки и фреймворки отлично подходят для решения огромного спектра задач, однако они не решают одну специфичную, но часто встречающуюся задачу: предоставление доступа к низкоуровневому API операционной системы и возможность интегрирования низкоуровневого процедурного кода Win32 API с высокоуровневыми абстракциями на основе этого API.

# 1 АНАЛИЗ ИСХОДНЫХ ДАННЫХ

Первоначальной задачей моего проекта является написание объектно-ориентированной обёртки над частью функционала Win32 API для организации удобного объектно-ориентированного построения GUI и широким использованием идиомы RAII при написании кода, для обеспечения его надёжности. Перед началом проектирования необходимо выполнить анализ исходных данных. Далее по порядку:

Актуальная версия операционной системы Windows является Windows 10 Fall creators update (build 1709).

Графические технологии для построения:

* GDI+: объектно-ориентированная обёртка над GDI API [10];
* Direct2D: объектно-ориентированная технология рендеринга двухмерных изображений на графическом процессоре [11]. Является частью экосистемы DirectX;
* DirectX 9-12: объектно-ориентирования технология, предоставляющая низкоуровневый доступ к ресурсам графического процессора для достижения максимально возможной эффективности видео рендеринга [12].

Для раскрытия всех возможностей API операционной системы Windows необходимо огромное количество времени, поэтому некоторые функциональные возможности придётся обойти стороной.

API DirectX – невероятно мощное решение, но на его изучение и эффективное внедрение уйдёт много времени. Что касается Direct2D – этот API по сложности не сильно далёк от старшего брата в лице DirectX.

GDI+ - простое решение, которое почти идеально подойдёт для базовой реализации, так как данная библиотека имеет достаточно простой интерфейс, чтобы не тратить много времени на его изучение.

Язык программирования: C++17 [13].

В рамках этого проекта необходимо разработать тестовое приложение, которое будет демонстрировать работу обёртки. Этим тестовым приложением будет программа «Windows explorer» - простой файловый обозреватель на базе данной обёртки для операционной системы Windows.

Необходимо, чтобы эта программа могла выполнять все свои функциональные обязанности в полной мере на любой операционной системе Windows, начиная с версии Windows Vista (6.0).

# 2 ПРОГРАММНОЕ ПРОЕКТИРОВАНИЕ

## 2.1 Проектирование «обёртки»

Любое приложение в операционной системе Windows, общающееся с пользователем с помощью графического интерфейса должно иметь окно. Именно сущность окна и должна стать основополагающей для построения всей остальной логики. Первичное визуальное представление окна представлено на рисунке 2.1.1.

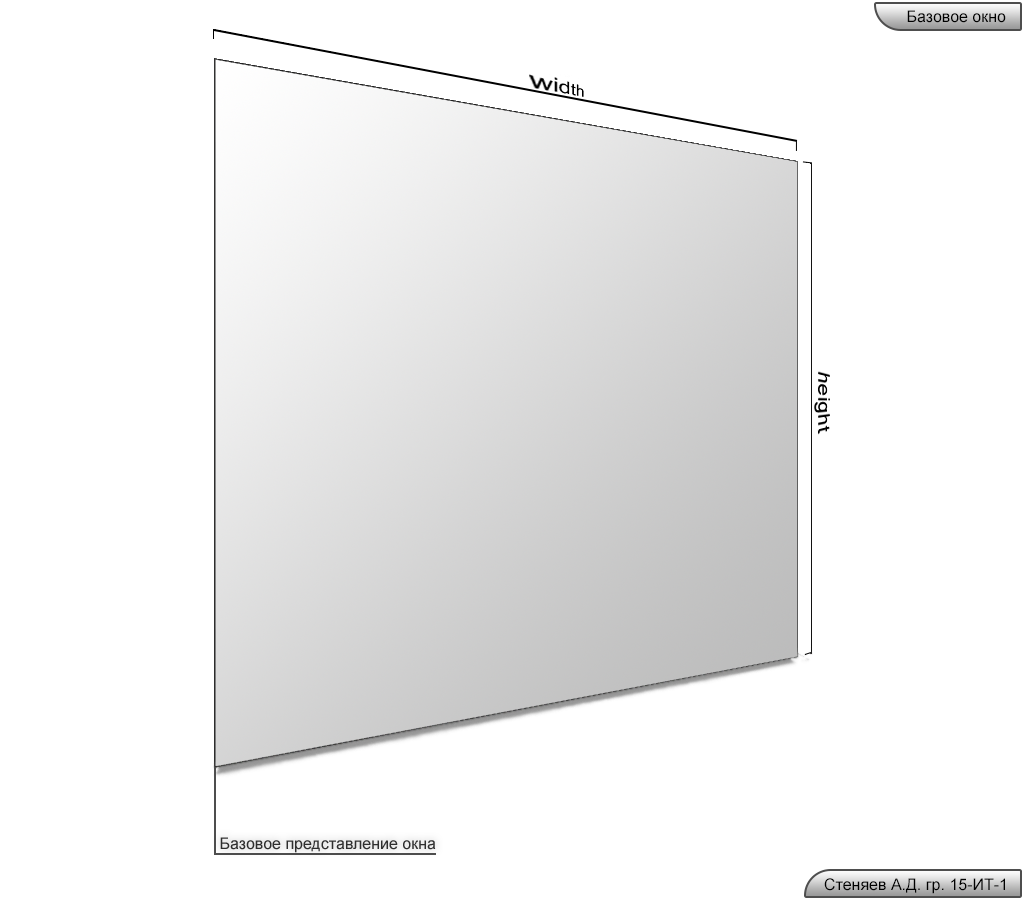


Рисунок 2.1.1 – базовое представление окна.

Для описания сущности окна введём класс «Окно», который будет описывать базовый функционал окна и инкапсулировать базовые обработчики от пользователя. Все последующие окна должны будут строится на основе базового класса «Окно». На рисунке 2.1.2 представлена диаграмма классов, расширяющих базовый класс «Окно».

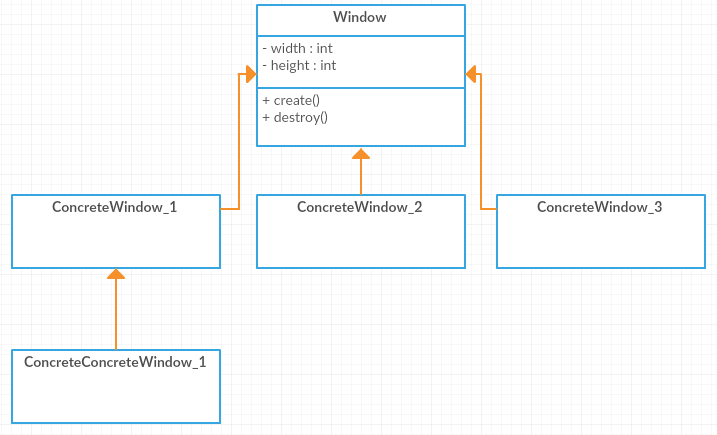


Рисунок 2.1.2 – диаграмма расширения базового класса.

Базовый класс должен включать в себя механизмы создания окна, уничтожения окна, обработки оконных событий и рассылки сообщений другим окнам об этих событиях, предоставления возможности манипулировать размерами окна, его положением и содержимым (изображения).

Создание любого окна начинается с регистрации класса окна. Чтобы каждый раз не создавать огромное количество классов окна, можно ограничиться одним, который будет общим для всех окон. Это значит, что перед регистрацией класса окна следует убедится в том, что этот класс ещё не зарегистрирован, чтобы зарегистрировать его и использовать, или чтобы не пытаться зарегистрировать его снова. Следующим шагом является создание самого окна вызовом системной процедуры. Тут есть некоторые моменты, требующие внимания: окно может иметь родителя, а может не иметь. Это значит, что при создании окна надо учитывать намерения пользователя сделать создаваемое окно дочерним, чтобы применить к создаваемому окну стили, подходящие для дочернего окна, иначе – для базового. Чтобы определить, является ли окно дочерним, достаточно при создании указать или не указать родителя, что приведёт к созданию дочернего и базового окон соответственно. На рисунке 2.1.3 показано, как располагаются окна «родитель» и «дочернее».

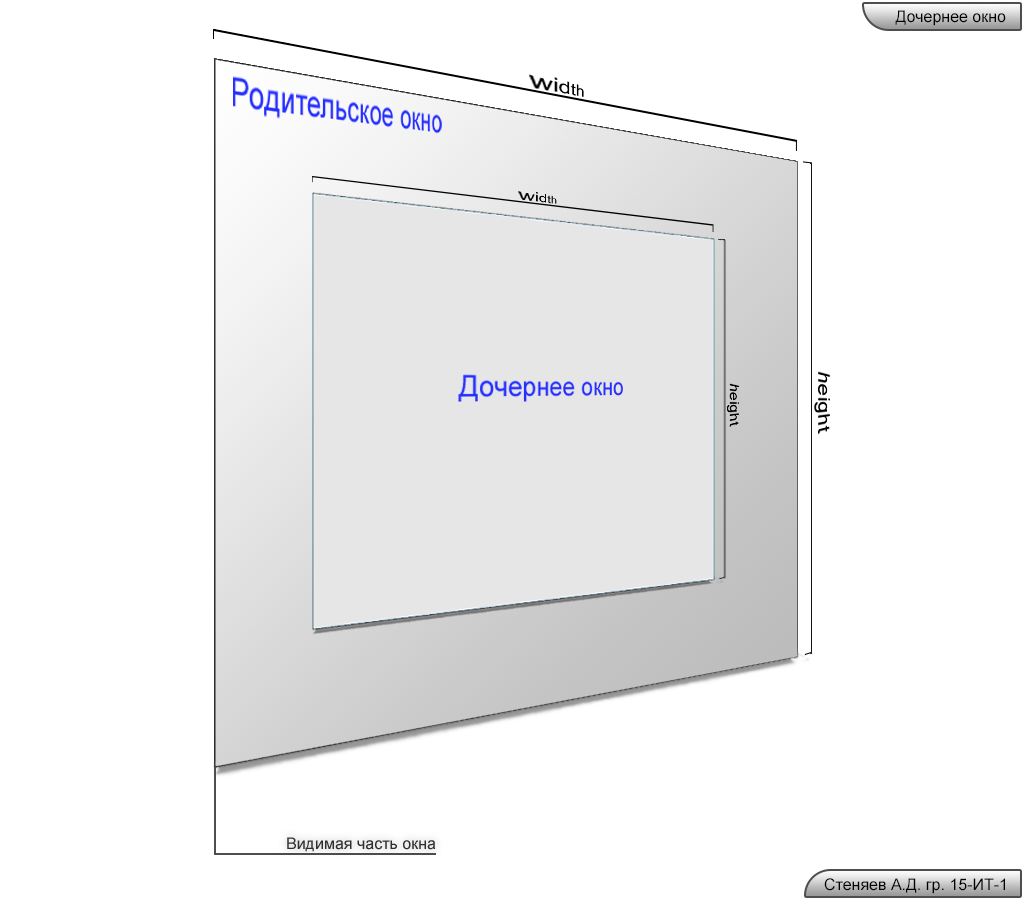


Рисунок 2.1.3 – расположение дочернего окна на поверхности родительского.

Класс «Окно» должен содержать в себе методы, которые возвращают текущее состояние окна и данные о нём. Важными параметрами являются размер и положение окна, которые пользователь может использовать в своих целях. Например, чтобы пользователь класса мог определить, где располагается окно и на основе этого определить какой размер окна ему нужен. Эта информация не редко используется, поэтому необходимо предоставить простой способ её получать. Так же важным параметром является «хэндл» окна. Все процедуры Win32 API, работающие с окнами, используют его для идентификации окна, в отношение которого выполняются действия. Поэтому необходимо реализовать доступ к этим данным, чтобы обеспечить возможность использовать данные объектов класса «Окно» и его производных с процедурами Win32 API.

Имя окна так же играет свою роль в операционной системе Windows. По этому имени пользователи могут различать свёрнутые окна в панели задач. Некоторые окна могут использовать своё имя для предоставления некоторых данных. Одним из таких окон в операционной системе Windows является Button. Поэтому следует дать пользователю возможность получить и установить имя окна в любой момент, когда ему это нужно.

Бывают моменты, когда необходимо узнать, где находится окно в пределах рабочего стола, а не в пределах окна. Для решения этой задачи так же можно предоставить метод, который все расчёты выполнит самостоятельно.

Так же необходимо добавить в класс «Окно» функционал, позволяющий манипулировать дочерними окнами без их оповещения. Данный функционал позволит, например, перемещать дочернее окно, которое должно всегда находится в нижнем правом углу. Без этого функционала пользователем придётся реализовывать его самостоятельно, что не очень удобно. К изменениям состояния дочерних окон относится изменение положения или размера окна при изменении размера окна-родителя. Так же этот функционал необходимо сделать настраиваемым, чтобы изменения дочерних окон происходили не во всех случаях, а только в определённых.

Границы окна – часть окна, которая позволяет манипулировать размерами этого окна. Это значит, что в классе «Окно» должны быть методы, позволяющие задавать размер и положение границ окна. Размер границ будет определять, какая часть любой из сторон окна может быть задействована для изменения размера этого окна. Так же бывают моменты, когда пользователь класса хочет, чтобы окно могло изменяться только в определённых направлениях. Для этого надо добавить метод, с помощью которого пользователь сможет установить стороны окна, участвующие в изменение его размера.

От размера окна зависит то, сколько всего может вместиться в его видимую часть. Однако иногда приходится ограничивать максимальный размер окна. Для этого необходимо добавить метод, который будет устанавливать максимальные размеры для окна.

Шапка стандартных окон в операционной системе Windows позволяет перемещать окна в любом направление. Это значит, что в класс «Окно» надо включить методы, включающие шапку и определяющие её размер. Это позволит пользователям окна перемещать окно в любое место, если в этом есть необходимость.

В процессе рисования сложных сцен (графики, составные картинки и прочее) и(или) их частой перерисовке начинает появляться неприятный эффект, называемый мерцанием. Это связанно с тем, что рисуемое изображение тут же отображается в видимой части окна, из-за чего пользователь видит весь процесс построения итогового изображения. Для предотвращения такого поведения используется достаточно простой метод – двойная буферизация. Суть метода заключается в том, чтобы рисовать сцену не в видимую часть окна, а в какую-то область памяти, из которой, при завершении рисования, перенести результат в видимую часть. Сам процесс копирования происходит намного быстрее, чем процесс рисования и отображение результата, поэтому метод является достаточно эффективным, однако он требует дополнительных ресурсов, что не является проблемой, так как в настоящие время современные компьютеры обладают достаточным запасом ресурсов, чтобы хранить в оперативной памяти несколько десятков таких буферов. В идеале размер буфера должен соответствовать размеру видимой области окна, чтобы обновлялась вся область видимой части, однако буфер может иметь и больший размер, что полезно в случае, когда надо отображать большое количество данных, которые не влезают в рамки видимой области: можно создать буфер достаточно размера, а затем выводить в видимую область окна нужную его часть.

Рисование должно производиться в три этапа:

* 1. Подготовка изображения к рисованию.
  2. Рисование изображения в временную память (буфер).
  3. Копирование содержимого буфера в видимую часть окна.

При подготовке изображения выполняются самые базовые действия:

* загрузка всех составных частей;
* заполнение буфера базовым цветом (очистка);
* расчёт позиций всех элементов на сцене;
* преобразования, выполняемые над растровыми изображениями (вращение, масштабирование, выделение рисуемой части и т.д.);
* установка регионов рисования.

На рисунке 2.1.4 показан пример составленной сцены, подготовленной к рисованию в буфер.



Рисунок 2.1.4 – подготовка к рисованию сцены в буфер.

Под загрузкой всех составных частей понимается чтение в оперативную память всех элементов будущей составной сцены, чтобы потом собрать их в нужном порядке и в нужных местах, а затем вывести.

Очистка буфера – заполнение всего буфера каким-то определённым цветом, чтобы потом поверх него что-то нарисовать. Это не обязательный шаг и применяется он только в том случае, если в сцене постоянно надо иметь монотонный задний фон без создания дополнительных инструментов для создания этого фона.

Расчёт позиций нужен для того чтобы заранее определить, где что будет располагаться, чтобы не выполнять лишние вызовы процедур и(или) методов.

После всех расчёт при необходимости можно выполнить операции трансформации растровых изображений, чтобы они заняли свои места на сцене.

Регионы рисования определяют простые или сложные области, в которых будет происходить рисование. С помощью регионов можно, например, ограничить видимую часть окна так, чтобы вместо прямоугольника была, например, звезда.

После подготовки начинается сам процесс рисования изображения. Каждая часть сцены последовательно рисуется в буфер. Пока идёт этот процесс, пользователь не видит изменений в видимой части окна, что не особо важно, если сцена простая, так как она рисуется достаточно быстро. После процесса рисования к буферу можно применить всевозможные пост эффекты: негатив, блюр, гауссово размытие и т.д. Так же результат рисования можно в дальнейшем использовать при рисовании других сцен, что, несомненно, очень удобно, так как достаточно всего раз нарисовать какую-то неизменную часть сцены, которая потом будет использоваться в других сценах.

Результат рисования сцены в буфер показан на рисунке 2.1.5.



Рисунок 2.1.5 – Результат рисования сцены в буфер.

Чтобы пользователь увидел результат рисования необходимо скопировать содержимое буфера в видимую часть окна. Делается это точно так же, как и в случае с копированием в другой буфер. У каждого окна есть свой экранный буфер, который и хранит информацию о цвете каждого пиксела, чтобы в любой момент восстановить её. Поэтому после копирования данных в буфер окна мы получаем видимое изображение в самом окне. Результат копирования изображения из буфера в видимую часть показан на рисунке 2.1.6.



Рисунок 2.1.6 – Результат копирования данных из буфера в видимую часть окна.

Для внедрения этого метода в проект необходимо написать класс, описывающий буфер и предоставляющий удобный интерфейс для доступа к данным из этого буфера:

1. Пользователь буфера должен иметь возможность получить доступ к данным этого буфера. В Win32 API буфер можно создать на основе карты битов (BITMAP), а для доступа к этой карте битов в процессе рисования нужен контекст устройства рисования (HDC). Из этого следует, что класс должен предоставлять доступ к своему контексту устройства, чтобы в буфер можно было рисовать, и к своему «хенделу» карты битов, чтобы из него в любой момент можно было получить всю необходимую информацию или использовать для более корыстных целей (разделение памяти между двумя буферами).
2. Буфер должен хранить информацию о своём размере, чтобы пользователь или другой буфер могли в любой момент узнать размер этого буфера.
3. Пользователю в любой момент может понадобится тот же буфер, но большего размера. Это значит, что буфер должен иметь возможность менять свой размер при необходимости, чтобы адаптироваться к новым требованиям пользоваться.
4. Буфер может существовать сколь угодно долго, а его содержимое может быть достаточно востребовано. Следовательно, необходимо реализовать возможность копирования этих данных в другие буферы (например, из буфера в видимую часть окна).

На рисунке 2.1.7 показана диаграмма вариантов использования класса «Буфер изображения».

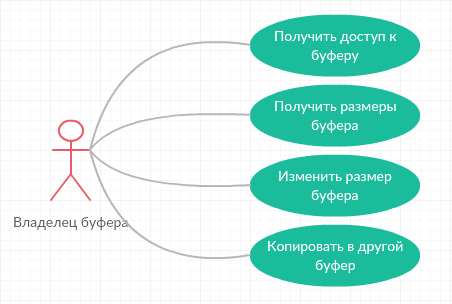


Рисунок 2.1.7 – диаграмма использования класса «Буфер изображения».

Для корректной работы программы окнам надо общаться. Для того чтобы наладить общение между окнами не надо создавать много методов и хранить ссылки на другие окна. Для организации общения между окнами следует использовать паттерн проектирования «Слушатель». Использование данного паттерна позволит наладить связь между окнами следующим образом: окно, желающее получать уведомления о каких-либо действиях от другого окна, должно оформить подписку на соответствующие обновления. При возникновении какого-то события все подписавшиеся на рассылку окна будут получать уведомления о том, что это событие произошло. На основе этого можно наладить стабильное общение между окнами. Единственная проблема, которая может возникнуть в процессе разработки: установка этих связей. Для её решения необходимо грамотно подойти к проектированию как самой библиотеки, так и итогового приложения, уложившись при этом в короткий срок.

Пример установки связи между окнами показан на рисунке 2.1.8.

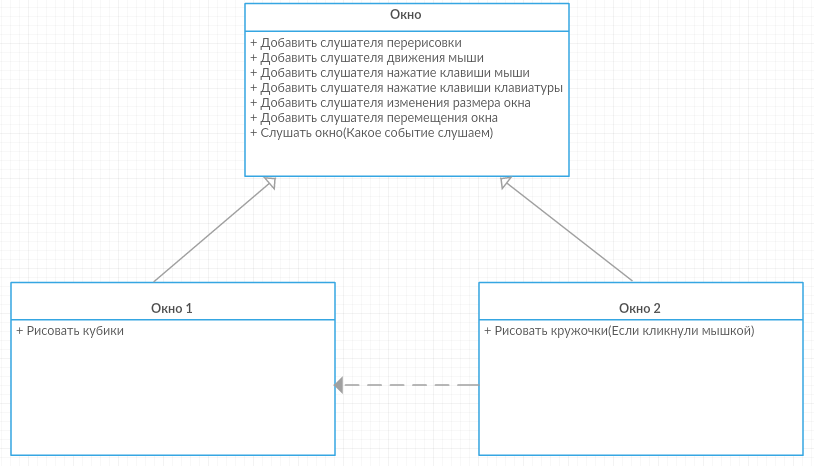


Рисунок 2.1.8 – Связь между окнами.

Некоторые данные и процедуры, которые являются общими для всех и могут потребоваться пользователю можно вынести в отдельный класс «System». К общим данным относятся данные о размерах экрана, данные о уровне яркости и звука, данные о аппаратном обеспечении и т.д. Класс System должен представлять единый интерфейс для получения всех необходимых системных данных. Тут всё просто: класс со статическими методами, которые возвращают какие-то системные данные в удобном для обработки виде.

Файлы играют большую роль в работе огромного количества приложений, поэтому необходимо реализовать базовый интерфейс для работы с файловой системой операционной системы Windows. Для этого нужно ввести класс «Файл», который будет описывать сущность того или иного файла, будь то реальный файл или папка. Так же необходимо реализовать возможность получать информацию о наличии логических и системных дисков, чтобы пользователь класса знал, куда он вообще может пойти.

Класс «Файл» должен обеспечить пользователю информацию о том, является ли сущность файлом или это каталог, упростить пользователю получение информации об имени файла (уметь выделять его из общего пути к файлу с его именем), получать директорию расположения файла из полного пути, директорию, которая шла до директории, в которой находится файл (для удобства), а предоставлять список всех файлов, которые располагаются в каталоге. Так же в структуру с информацией о файле можно включить информацию о том, есть ли у него иконка и, если есть, информацию о самой иконке.

## 2.2 Проектирование тестового приложения

Тестовым приложением будет файловый обозреватель, построенный на основе спроектированной «обёртки» над Win32 API. Данное приложение будет отображать список файлов в папках, позволит их запускать, а также позволит перемещаться между папками и логическими дисками.

Так как в проектирование «обёртки» был спроектирован пока что только каркас, то отталкиваться надо именно от него. Для начала необходимо определить то, как будет выглядеть интерфейс приложения. Для этих частей интерфейса надо будет реализовать функционал, соответствующий назначению каждой из частей интерфейса.

Для начала необходимо создать главное окно для приложения. Это окно будет содержать в себе все элементы управления приложения, а также отвечать за масштабирование и перемещение. Далее следует сделать окна, которые будут отвечать за управление главным окном, а именно:

* кнопка закрытия главного окна;
* кнопка масштабирования главного окна;
* кнопка сворачивания главного окна.

Кнопка закрытия окна будет отвечать за то, чтобы главное окно могло закрыться, а приложение могло завершить свою работу.

Кнопка масштабирования будет отвечать за изменение размеров окна от исходного размера к максимальному и наоборот.

Кнопка сворачивания окна будет отвечать за сворачивание окна.

Главной частью тестового приложения будет список с файлами. Этот список будет располагаться по центру и будет отвечать за следующие действия:

* отображение списка файлов;
* выбор любого файла из списка;
* переходы между каталогами и логическими дисками;
* дополнительные операции, связанные с файлами.

Чтобы отобразить список файлов текущего каталога окно будет запрашивать у класса «Файл» список файлов в текущем каталоге окна. После этого список будет сохранятся в хранилище окна и использоваться для его вывода на экран.

Выбор файлов происходит следующим образом: пользователь водит мышкой, на что реагирует окно и определяет, где в данный момент находится курсор. Если курсор находится над строкой с названием файла, то соответствующая строка выделяется. Если пользователь делает клик мыши, то эта строка подсвечивается цветом, обозначающим выделение строки. Если пользователь делает двойной клик по строке, то окно начинает выяснять, чем является файл – папкой или файлом. Если пользователь сделал двойной клик по папке, то окно делает запрос классу «Файл» с целью получить список файлов в новой директории, после чего обновляет свои данные и выводит на экран новый список файлов. Если пользователь сделал двойной клик по файлу, то этот файл должен запуститься, если это исполняемый файл, или открыться в программе по умолчанию. Эти решения нужно передать под контроль операционной системе.

Чтобы пользователю было удобно перемещаться между каталогами и логическими дисками надо добавить три дополнительные кнопки:

* кнопка возврата на один уровень выше;
* кнопка возврата в предыдущий каталог;
* кнопка возврата в предыдущий каталог, из которого был сделан шаг назад.

Окно со списком файлов должно будет подписаться на эти окна и получать уведомления о их нажатии. При нажатии на кнопку «на один уровень выше» будет выполнен переход на один каталог в иерархии выше, то есть если было «C:\Program Files\», то в результате текущим каталогом должен стать каталог «C:\». Кнопка возврата в предыдущий каталог будет выполнять переход из текущего каталога в каталог, из которого был выполнен переход в текущий. В отличии от кнопки для перехода на уровень выше эта кнопка будет возвращать в каталог, из которого был выполнен переход в текущий. Последняя кнопка «кнопка возврата в предыдущий каталог, из которого был сделан шаг назад» будет выполнять функцию возврата в каталог, из которого был сделан возврат. Иначе говоря, это повторение шага, который был сделан до того, как была выполнена операция возврата в предыдущий каталог.

Примерный внешний вид приложения представлен на рисунке 2.2.1.

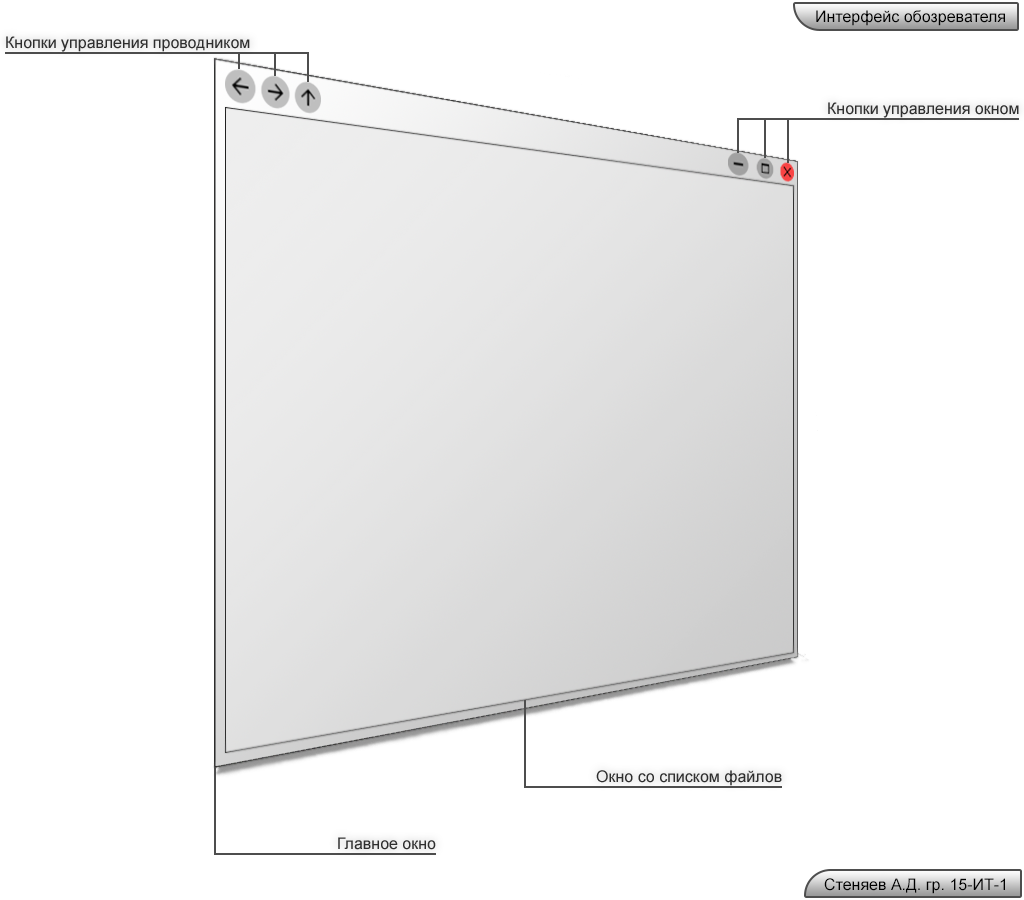


Рисунок 2.2.1 – Примерный внешний вид тестового приложения

# 3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

## 3.1 Детальная реализация функциональных частей ПО

### 3.1.1 Детальная реализация «обёртки»

По проекту следует начать с реализации главного класса окна. После его объявление следует его заполнить полями, которые приведены в листинге 1.

**Листинг 1** – Поля класса Window

1. **private**:
2. **static** std::**map**<**HWND**, **Window**\*> s\_windowsMap;
3. **static** std::**wstring** \_className;
4. **static** **ULONG\_PTR** \_gdiplusToken;
5. **static** **const** **UINT** TIMER\_UPP\_HOVER = 10000;
7. **Window**\* \_parent;
8. std::**list**<Window\*> \_childList;
10. std::**list**<MouseClickHandler> \_mouseClickHandlers;
11. std::**list**<MouseWheelHandler> \_mouseWheelHandlers;
12. std::**list**<MouseMoveHandler> \_mouseMoveHandlers;
13. std::**list**<KeyboardHandler> \_keyboardHandlers;
14. std::**list**<ParentHandler> \_parentHandlers;
15. std::**list**<PaintHandler> \_paintHandlers;
16. std::**list**<TimerHandler> \_timerHandlers;
17. std::**list**<HoverHandler> \_hoverHandlers;
19. **int** \_width, \_hieght, \_oldWidth, \_oldHieght;
20. **int** \_minWidth, \_minHieght;
21. **int** \_maxWidth, \_maxHieght;
22. **int** \_pos\_x, \_pos\_y;
23. **int** \_g\_pos\_X, \_g\_pos\_Y;
24. **int** \_borderSize;
25. **HWND** \_hWnd;
27. **WNDCLASSEX** \_WndClass;
28. std::**wstring** \_windowName;
30. **bool** \_resizing;
31. **bool** \_redrawWhereResizing;
32. **bool** \_thisWindowIsCreated;
33. **bool** \_hoverStatus;
34. **bool** \_moveWhenParentResiz\_X, \_moveWhenParentResiz\_Y;
35. **bool** \_resizeWhenParentResize\_Width, \_resizeWhenParentResize\_Height;
36. **bool** \_canBeResize\_top, \_canBeResize\_bottom;
37. **bool** \_canBeResize\_left, \_canBeResize\_right;
38. **bool** \_haveHeader;
39. **bool** \_isLocked;
41. std::**shared\_ptr**<RenderBuffer> \_renderBuffer;
42. **int** \_scrollbarHorizontalStatus;
43. **int** \_scrollbarVerticalStatus;
44. **int** \_scrollbarHorizontalStepSize;
45. **int** \_scrollbarVerticalStepSize;
46. **bool** \_scrollbarHorizontal\_IsEnable;
47. **bool** \_scrollbarVertical\_IsEnable;

Поле s\_windowsMap во второй строке листинга 1 является картой окон класса Window. С помощью этой карты определяется окно, которому было адресовано сообщение. Окна добавляются в карту при создании и удаляются при удалении.

Поле \_className в третьей строке листинга 1 хранит имя класса окна, что в пользователь мог получить и использовать в своих целях.

Поле \_gdiplusToken в четвёртой строке листинга 1 является указателем на объект GDI+, который будет использоваться для рисования графических элементов.

Поля \_parent и \_childList в седьмой и восьмой строках листинга 1 хранят адреса родительских и дочерних окон. Если у окна нет родительского окна, то поле \_parent будет иметь значение nullptr, в противном случае – адрес окна-родителя. Если у окна есть дочерние окна, то поле \_childList, являющиеся списком, будет содержать адреса всех окон, являющихся дочерними окнами конкретного окна.

В 10-17 строках листинга 1 определены поля, являющиеся списками адресов методов или процедур, вызываемых в случае возникновения конкретного события. В эти списки добавляются методы и процедуры, в которых пользователь будет обрабатывать события.

В 19 строке листинга 1 определены поля \_width и \_hieght, которые хранят текущий размер окна, а так же \_oldWidth и \_oldHieght, которые хранят предыдущие размеры окна.

В строках 20 и 21 листинга 1 определены поля, хранящие значения максимального и минимального размеров окна. Эти параметры будут использоваться для определения минимального и максимального размеров окна.

В строках 22 и 23 листинга 1 определены поля, хранящие значения локального и глобального положений окна.

В строке 24 листинга 1 определена переменная \_borderSize, хранящая значение размера границ окна. Это значение используется для активных границ, чтобы определить интервал между краем окна и некоторой координатой на оси, перпендикулярной исследуемой стороне, в пределах которого пользователь может начать процесс изменения размера окна.

В строке 25 листинга 1 определена переменная \_hWnd, хранящая «хэндл» созданного окна. Хранение этой переменной позволит получить к ней доступ при необходимости.

Поле \_WndClass, находящееся на 27 строке листинга 1, хранит информацию о классе окна. Под классом тут понимается структура, хранящая данные, необходимые для создания окна, такие как, например, стили класса (перерисовка, общий контекст рисования и т.д.).

В поле \_windowName, находящееся на 28 строке листинга 1, хранится имя окна.

В строках с 30 по 39 листинга 1 определены поля, которые определяют, активна ли та или иная особенность окна. Поле \_resizing является логической переменной, которая истина, если окно в данный момент меняет размер. Поле \_redrawWhereResizing является логической переменной, в случае истинности которой будет перерисовываться окно при изменении его размеров. Поле \_thisWindowIsCreated служит флагом, запрещающим повторное создание окна, если оно уже было создано. Поле \_hoverStatus хранит истинное значение в случае, когда курсор находится над окном. Наличие этого поля, например, поможет создать элемент управления «кнопка». Поля \_moveWhenParentResiz\_X и \_moveWhenParentResiz\_Y определяют, будет ли окно двигаться по оси OX и(или) по оси ОY в случае, когда родительское окно меняет свой размер и эти поля имеют истинное значение. Поля \_resizeWhenParentResize\_Width и \_resizeWhenParentResize\_Height определяют, будет ли окно изменять свои вертикальный и горизонтальный размеры в случае, когда их родительское окно меняет свой размер и эти поля имеют истинное значение. Поля \_canBeResize\_top и \_canBeResize\_bottom определяют возможность изменения размеров окна по вертикали (вверх и вниз соответственно). Поля \_canBeResize\_left и \_canBeResize\_right определяют возможность изменения размеров окна по горизонтали (влево и вправо соответственно). Поле \_haveHeader определяет, есть ли окна «шапка», позволяющая перемещать это окно. Поле \_isLocked определяет, заблокировано ли окно в текущий момент.

Поле \_renderBuffer, находящееся на строке 41 листинга 1, является «умным» указателем на объект класса RenderBuffer, который будет описан позже. Указатель используется из-за необходимости создавать буфер до того, как будет создано само окно.

Поля, начиная со строки 42 и заканчивая строкой 47 листинга 1, отвечают за возможность «скроллить» содержимое окна.

Все эти поля используются в методах класса. Все эти методы будут описаны далее.

При создании объекта окна в первую очередь вызывается конструктор класса Window, код которого приведён в листинге 2.

**Листинг 2** – конструктор класса Window

1. Window::Window() :
2. \_width(0), \_hieght(0), \_oldWidth(0), \_oldHieght(0),
3. \_minWidth(50), \_minHieght(15),
4. \_pos\_x(0), \_pos\_y(0),
5. \_borderSize(3),
6. \_parent(nullptr),
7. \_thisWindowIsCreated(**false**),
8. \_moveWhenParentResiz\_X(**false**), \_moveWhenParentResiz\_Y(**false**),
9. \_resizeWhenParentResize\_Width(**false**),
10. \_resizeWhenParentResize\_Height(**false**),
11. \_canBeResize\_top(**false**), \_canBeResize\_bottom(**false**),
12. \_canBeResize\_left(**false**), \_canBeResize\_right(**false**),
14. \_resizing(**false**),
15. \_redrawWhereResizing(**true**),
17. \_scrollbarHorizontalStatus(0),
18. \_scrollbarVerticalStatus(0),
19. \_scrollbarHorizontalStepSize(0),
20. \_scrollbarVerticalStepSize(0),
22. \_haveHeader(**false**)
23. {
24. Gdiplus::**GdiplusStartupInput** gdiplusStartupInput;
25. Gdiplus::GdiplusStartup(&\_gdiplusToken,
26. &gdiplusStartupInput, NULL);
27. m\_registerTimerHendler(METHOD(&**Window**::timerCheckHoverWindow));
28. registerHendler(
29. METHOD(&**Window**::abstratcWindowScrollIventHandler));
30. }

В начале кода, представленного в листинге 2, начиная со второй строки и заканчивая двадцать первой строкой, идёт инициализация полей класса значениями по умолчанию. Далее идёт тело конструктора, где в строках 24 и 25 идёт инициализация GDI+, в строке 25 вызывается метод, регистрирующий обработчик событий базового таймера, а в строке 26 регистрируется обработчик события вращения колёсика мыши. Регистрация обработчика базового таймера необходима для того чтобы постоянно проверять, находится ли курсор мыши над окном. Регистрация базового обработчика вращения колёсика мыши выполняется для того чтобы при его вращении в первую очередь выполнялись базовые действия, а уж потом все пользовательские. Это решение спорное и может быть в будущем изменено.

В тот момент, когда объект окна прекращает своё существование, вызывается его деструктор. Код деструктора приведён в листинге 3.

**Листинг 3** – деструктор класса Window

1. Window::~Window()
2. {
3. s\_windowsMap.erase(\_hWnd);
4. DestroyWindow(\_hWnd);
6. **if** (\_parent) {
7. \_parent->m\_removeChildWindow(**this**);
8. }
10. **if** (s\_windowsMap.size() == 0) {
11. Gdiplus::GdiplusShutdown(\_gdiplusToken);
12. PostQuitMessage(0);
13. }
14. }

Деструктор выполняет функцию освобождения используемых окном ресурсов, а также вызывает методы, которые удалят окно из всех списков и уничтожат его.

Далее рассмотрим методы класса Window. Все методы, связанные с получением и установкой каких-то значений, не требующих каких-либо дополнительных действий (расчёты и т.д.), будут опущены и доступны для изучения в файле с кодом в приложенном диске.

После создания объекта окна надо создать само окно. Это делается с помощью вызова метода create. Реализация метода create приведена в листинге 4.

**Листинг 4** – реализация метода create

1. **bool** Window::create(std::wstring& name,
2. Window& parent,
3. **int** pos\_x, **int** pos\_y,
4. **int** width, **int** hieght,
5. **bool** show)
6. {
7. **if** (\_thisWindowIsCreated) {
8. **return** **false**;
9. }
11. \_windowName = name;
12. \_pos\_x = pos\_x;
13. \_pos\_y = pos\_y;
14. \_oldWidth = \_width = width;
15. \_oldHieght = \_hieght = hieght;

18. **if** (!m\_create(&parent, show)) {
19. **return** **false**;
20. }
22. \_parent = &parent;
23. parent.m\_addChildWindow(**this**);
25. \_thisWindowIsCreated = **true**;
26. **return** **true**;
27. }

Метод, приведённый в листинге 4 принимает на вход имя окна, окно-родитель, координаты X и Y, ширину и высоту окна, а также логическую переменную, от которой зависит, будет ли окно изначально видно или не будет. В теле метода вначале выполняется проверка переменной, сигнализирующей о том, что окно уже создано, на истинность. Если окно уже создано, то новое мы не создаём и возвращаем ложный результат. В противном случае мы присваиваем окну переданные параметры, после чего вызываем метод m\_create, в котором начинается процесс создания окна. Определение метода m\_create приведено в листинге 5. После создания окна в качестве родителя устанавливается переданное окно, когда самому родителю в список детей записывается создаваемое окно. Устанавливаем флаг, указывающий на то, что окно уже создано, и возвращаем истинное значение.

**Листинг 5** – определение метода m\_create

1. **bool** Window::m\_create(Window\* parent, **bool** show)
2. {
3. **try** {
4. m\_registerClass();
5. m\_createWindow(parent);
6. }
7. **catch** (WindowClassException ex) {
8. ex.showMsg();
9. **return** **false**;
10. }
11. **catch** (WindowException ex) {
12. ex.showMsg();
13. **return** **false**;
14. }
15. **catch** (...) {
16. **return** **false**;
17. }
19. \_g\_pos\_X = getGlobalPosX();
20. \_g\_pos\_Y = getGlobalPosY();
22. SetWindowPos(\_hWnd, 0, 0, 0, 0, 0,
23. SWP\_NOSIZE
24. | SWP\_NOMOVE
25. | SWP\_FRAMECHANGED
26. | (show ? SWP\_SHOWWINDOW : SWP\_HIDEWINDOW));
27. **return** **true**;
28. }

Метод на вход принимает указатель на окно-родитель и параметр отображения, в случае истинности которого окно будет видимо, иначе - нет. Далее идёт попытка зарегистрировать класс окна с помощью вызова метода m\_registerClass, после чего вызывается метод m\_createWindow, который создаст окно и, в случае неудачи, выбросит исключение. Определение метода m\_registerClass приведено в листинге 6. Определение метода m\_createWindow приведено в листинге 7.

**Листинг 6** – определение метода m\_registerClass

1. **ATOM** Window::m\_registerClass() {
2. WNDCLASS wndclass;
3. **if** (GetClassInfo(GetModuleHandle(0), \_className.c\_str(), &wndclass)) {
4. **return** 1;
5. }
7. \_WndClass.cbSize = **sizeof**(\_WndClass);
8. \_WndClass.style = CS\_DBLCLKS | CS\_SAVEBITS;
9. \_WndClass.lpfnWndProc = WndProc;
10. \_WndClass.cbClsExtra = 0;
11. \_WndClass.cbWndExtra = 0;
12. \_WndClass.hInstance = GetModuleHandle(NULL);
13. \_WndClass.hIcon = LoadIcon(NULL, IDI\_APPLICATION);
14. \_WndClass.hCursor = LoadCursor(NULL, IDC\_ARROW);
15. \_WndClass.hbrBackground = (**HBRUSH**)GetStockObject(0);
16. \_WndClass.lpszMenuName = NULL;
17. \_WndClass.lpszClassName = \_className.c\_str();
18. \_WndClass.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);
20. **if** (!RegisterClassEx(&\_WndClass)) {
21. **throw** WindowClassException();
22. }
23. **return** 1;
24. }

**Листинг 7** – определение метода m\_createWindow

1. **bool** Window::m\_createWindow(Window\* parent)
2. {
3. **long** **int** style = ((parent) ? (WS\_CHILD | WS\_CLIPSIBLINGS) :
4. (WS\_POPUP | WS\_BORDER)) | WS\_CLIPCHILDREN | WS\_VISIBLE;
6. \_hWnd = CreateWindowExW(
7. 0,
8. \_className.c\_str(),
9. \_windowName.c\_str(),
10. style,
11. \_pos\_x,
12. \_pos\_y,
13. \_width,
14. \_hieght,
15. ((parent) ? (parent->getHWND()) : (nullptr)),
16. nullptr,
17. GetModuleHandle(0),
18. nullptr
19. );
21. **if** (!\_hWnd) {
22. **throw** WindowException(
23. L"Create Window "
24. + \_windowName
25. + L" error!"
26. );
27. }
28. s\_windowsMap.insert(std::pair<**HWND**, Window\*>(\_hWnd, **this**));
29. \_renderBuffer = std::make\_shared<RenderBuffer>(\_hWnd, \_width,\_hieght);
30. SendMessage(\_hWnd, WM\_CREATE, 0, 0);
31. **return** **true**;
32. }

Метод на вход принимает указатель на окно-родитель и параметр отображения, в случае истинности которого окно будет видимо, иначе - нет. Далее идёт попытка зарегистрировать класс окна с помощью вызова метода m\_registerClass, после чего вызывается метод m\_createWindow, который создаст окно и, в случае неудачи, выбросит исключение.

После создания окна начинается обработка сообщений, посланных этим окнам. Процедура, обрабатывающая оконные события указывается в структуре класса при регистрации класса окна, в случае класса Window вместо процедуры будет статический метод. Первым делом определяется, зарегистрировано ли окно в карте окон (листинг 1, строка 2), после чего, в случае успеха, начинается рассылка уведомлений и обработка, иначе вызывается обработчик по умолчанию, предоставленный в Win32 API. Код определения окна приведён в листинге 8.

**Листинг 8** – Поиск окна, которому адресовано сообщение

1. **if** (s\_windowsMap.find(hWnd) != s\_windowsMap.end()) {
2. Window\* window = s\_windowsMap[hWnd];
3. **switch** (msg) {
4. ...
5. **default**: **return** DefWindowProc(hWnd, msg, wParam, lParam);
6. }
7. **return** 0;
8. }
9. **return** DefWindowProc(hWnd, msg, wParam, lParam);

В строке 1 листинга 8 идёт поиск объекта окна по его «хенделу», почле чего, в случае успеха, получается указатель на объект окна и начинается обработка сообщений. В случае, если окно не найдено или не найден обработчик, то выполняется обработка по умолчанию.

В каждом обработчике вызывается метод, который отвечает за обработку конкретного события. Все эти методы имеют модификатор доступа private, поэтому доступ к ним можно получить только из самого класса и из дружественных классов.

Одним из сообщений, которое надо обрабатывать, является сообщение WM\_GETMINMAXINFO. Оно отправляется окну тогда, когда окно пытается переместиться или изменить размер. Обработчик этого события приведён в листинге 9.

**Листинг 9** – Обработчик события WM\_GETMINMAXINFO

1. **void** Window::m\_WndProcHandler\_GetMinMaxInfo(
2. Window\* wnd,
3. **HWND** hWnd,
4. **UINT** msg,
5. **WPARAM** wParam,
6. **LPARAM** lParam)
7. {
8. MINMAXINFO\* min\_max = **reinterpret\_cast**<MINMAXINFO\*>(lParam);
10. min\_max->ptMinTrackSize.x = wnd->\_minWidth;
11. min\_max->ptMinTrackSize.y = wnd->\_minHieght;
12. min\_max->ptMaxTrackSize.x = wnd->\_maxWidth;
13. min\_max->ptMaxTrackSize.y = wnd->\_maxHieght;
14. }

В первых 6 строках обработчика в листинге 9 указано имя обработчика и его параметры. Все остальные обработчики имеют такую же структуру. В 8 строке листинга 9 идёт приведение параметра lParam в MINMAXINFO\*, так как в параметре lParam передаётся указатель именно на эту структуру. Далее в полях этой структуры устанавливаются минимальный и максимальный размеры окна.

В процессе изменения размера окна приходит сообщение WM\_SIZING. Оно приходит всё время, пока меняется размер окна. Код обработчика этого сообщения приведён в листинге 10.

**Листинг 10** – Обработчик события WM\_SIZING

1. **void** Window::m\_WndProcHandler\_Sizing(
2. Window\* wnd,
3. **HWND** hWnd,
4. **UINT** msg,
5. **WPARAM** wParam,
6. **LPARAM** lParam)
7. {
8. RECT\* rect = **reinterpret\_cast**<RECT\*>(lParam);
10. wnd->\_oldWidth = wnd->\_width;
11. wnd->\_oldHieght = wnd->\_hieght;
12. wnd->\_width = rect->right - rect->left;
13. wnd->\_hieght = rect->bottom - rect->top;
15. wnd->m\_calculateNewPositionWindowIfParentResize();
16. wnd->m\_calculateNewSizeWindowIfParentResize();
17. }

В строке 8 листинга 10 выполняется приведение из типа LPARAM в тип RECT\*, так как в параметре lParam передаётся указатель на структуру типа RECT. После этого в поля, хранящие старый размер окна, помещают значения переменных, хранящих значения текущего размера, а в переменные с текущим размером записываются данные о новом размере окна. После этого вызываются два метода, которые рассчитывают новые положения и размеры дочерних окон. Коды этих методов приведены в листингах 11 и 12.

**Листинг 11** – Метод изменения положений дочерних окон

1. **void** Window::m\_calculateNewPositionWindowIfParentResize()
2. {
3. **for** (Window\* window : \_childList) {
4. **if** (window->\_moveWhenParentResiz\_X
5. || window->\_moveWhenParentResiz\_Y) {
6. **int** childPos\_x = window->\_pos\_x;
7. **int** childPos\_y = window->\_pos\_y;
9. **int** newChildPos\_x = (window->\_moveWhenParentResiz\_X) ?
10. (\_width - (\_oldWidth - childPos\_x)) :
11. (childPos\_x);
12. **int** newChildPos\_y = (window->\_moveWhenParentResiz\_Y) ?
13. (\_hieght - (\_oldHieght - childPos\_y)) :
14. (childPos\_y);
16. window->moveWindowPos(newChildPos\_x, newChildPos\_y, **false**);
17. }
18. }
19. }

**Листинг 12** – Метод изменения размеров дочерних окон

1. **void** Window::m\_calculateNewSizeWindowIfParentResize()
2. {
3. **for** (Window\* window : \_childList) {
4. **if** (window->\_resizeWhenParentResize\_Width
5. || window->\_resizeWhenParentResize\_Height) {
6. **int** childWidth = window->\_width;
7. **int** childHeight = window->\_hieght;
9. **int** newChildWidth = (window-> \_resizeWhenParentResize\_Width) ?
10. (childWidth + \_width - \_oldWidth) :
11. (childWidth);
12. **int** newChildHeight = (window-> \_resizeWhenParentResize\_Height) ?
13. (childHeight + \_hieght - \_oldHieght) :
14. (childHeight);
16. window->resizeWindow(newChildWidth, newChildHeight, **true**);
17. }
18. }
19. }

В третьей строке листингов 11 и 12 выполняется обход списка дочерних окон. Для каждого окна проверяется условие, нужно ли ему менять положение или размер в случае изменения размера родителя. Если нужно, то выполняются все расчёты, связанные с определением нового положения и(или) нового размера окна, после чего изменение применяется к окну.

Одним из наиболее часто обрабатываемых сообщений является сообщение WM\_PAINT. Это сообщение приходит окну всякий раз, когда необходимо выполнить его перерисовку. Код обработчика этого сообщения приведён в листинге 13.

**Листинг 13** – Обработчик сообщения WM\_PAINT

1. **void** Window::m\_WndProcHandler\_Paint(
2. Window\* wnd,
3. **HWND** hWnd,
4. **UINT** msg,
5. **WPARAM** wParam,
6. **LPARAM** lParam)
7. {
8. PAINTSTRUCT ps;
9. **HDC** hDC = BeginPaint(wnd->\_hWnd, &ps);
10. **HDC** hDC\_Buffer = wnd->\_renderBuffer->getDC();
12. Gdiplus::Graphics graphics(hDC\_Buffer);
14. **for** (auto handler : wnd->\_paintHandlers) {
15. handler(graphics);
16. }
18. wnd->\_renderBuffer->copyTo(
19. hDC,
20. wnd->\_scrollbarHorizontalStatus,
21. wnd->\_scrollbarVerticalStatus,
22. 0, 0,
23. wnd->\_width, wnd->\_hieght
24. );
26. EndPaint(wnd->\_hWnd, &ps);
27. }

В восьмой строке листинга 13 объявляется переменная типа PAINTSTRUCT, которая будет содержать в себе информацию о том, какая честь перерисовывается и т.д. Далее, в строке 9 листинга 13, вызывается процедура BeginPaint, которая возвращает контекст устройства, в котором будет происходить процесс рисования. Однако рисовать мы будем не в этот контекст, так как это может вызывать неприятные мерцания при рисование сложной сцены. Вместо этого в строке 10 листинга 13 мы получаем контекст буфера, в котором и будет строится изображение. В строке 12 листинга 13 создаётся объект GDI+, который будет отвечать за рисование. В конструктор объекта передаётся контекст устройства буфера, что означает, что всё рисование будет происходить в буфере. Далее вызываются все пользовательские обработчики рисования, в которых происходит рисование. После того, как всё нарисовано, в строке 18 листинга 13 вызывается метод буфера copyTo, который копирует определённую часть буфера в видимую часть окна. Стоит заметить, что размер копируемой части равен размеру видимой части окна.

Во время перемещения окно получает уведомление WM\_MOVE. Код обработчика этого сообщения приведён в листинге 14.

**Листинг 14** – Обработчик сообщения WM\_MOVE

1. **void** Window::m\_WndProcHandler\_Move(
2. Window\* wnd,
3. **HWND** hWnd,
4. **UINT** msg,
5. **WPARAM** wParam,
6. **LPARAM** lParam)
7. {
8. **int** oldPosX = wnd->\_pos\_x;
9. **int** oldPosY = wnd->\_pos\_y;
10. wnd->\_pos\_x = LOWORD(lParam);
11. wnd->\_pos\_y = HIWORD(lParam);
13. wnd->eventMoveWindow(oldPosX, oldPosY);
15. ParentEvent parentEvent;
16. parentEvent.Code = PARENT\_MOVE;
17. parentEvent.Pos\_X = wnd->\_pos\_x;
18. parentEvent.Pos\_Y = wnd->\_pos\_y;
19. parentEvent.Width = wnd->\_width;
20. parentEvent.Height = wnd->\_hieght;
22. **for** (auto child : wnd->\_childList) {
23. **for** (auto handler : child->\_parentHandlers) {
24. handler(parentEvent);
25. }
26. }
27. }

В восьмой и девятой строках листинга 14 объявляются переменные, которые запоминаю предыдущее положение окна. В десятой и одиннадцатой и девятой строках листинга 14 полям, хранящим значения положения окна, присваиваются значения нового положения окна. Затем вызывается метод окна eventMoveWindow, который пользователи класса могут перегрузить и в нём определить свои действия, после чего идёт заполнение структуры события ParentEvent, которая рассылается в обработчики всем дочерним окнам, которые подписались на получение информации о событиях их родителя (а вдруг это важно).

Вовремя изменение размера окна приходит сообщение WM\_SIZE. В отличии от WM\_SIZING это сообщение приходит не так часто, и оно пригодно для обработки пользователем. Код обработчика приведён в листинге 15.

**Листинг 15** – Обработчик сообщения WM\_SIZE

1. **void** Window::m\_WndProcHandler\_Size(
2. Window\* wnd,
3. **HWND** hWnd,
4. **UINT** msg,
5. **WPARAM** wParam,
6. **LPARAM** lParam)
7. {
8. ParentEvent parentEvent;
9. parentEvent.Code = PARENT\_RESIZE;
10. parentEvent.Pos\_X = wnd->\_pos\_x;
11. parentEvent.Pos\_Y = wnd->\_pos\_y;
12. parentEvent.Width = wnd->\_width;
13. parentEvent.Height = wnd->\_hieght;
15. **for** (auto child : wnd->\_childList) {
16. **for** (auto handler : child->\_parentHandlers) {
17. handler(parentEvent);
18. }
19. }
21. wnd->eventSizeWindow(wnd->\_oldWidth, wnd->\_oldHieght);
22. **if** (wnd->\_redrawWhereResizing || !wnd->\_resizing) {
23. RedrawWindow(hWnd, NULL, NULL,
24. RDW\_INVALIDATE | RDW\_NOERASE
25. | RDW\_INTERNALPAINT | RDW\_UPDATENOW | RDW\_ALLCHILDREN);
26. }
27. }

Тут почти всё тоже самое, что и в случае с перемещением окна, только ещё дополнительно решается вопрос о перерисовке окна.

Когда пользователь водит курсор мыши то, если курсор захвачен окном или он находится над активным окном, окну приходит сообщение WM\_MOUSEMOVE. Код обработчика этого события приведён в листинге 16.

**Листинг 16** – Обработчик сообщения WM\_MOUSEMOVE

1. **void** Window::m\_WndProcHandler\_MouseMove(
2. Window\* wnd,
3. **HWND** hWnd,
4. **UINT** msg,
5. **WPARAM** wParam,
6. **LPARAM** lParam)
7. {
8. POINT cursorPoint;
9. GetCursorPos(&cursorPoint);
11. MouseEvent mouseEvent(LOWORD(lParam), HIWORD(lParam), cursorPoint.x, cursorPoint.y);
13. **for** (auto handler : wnd->\_mouseMoveHandlers) {
14. handler(mouseEvent);
15. }
16. }

В строках восемь и девять листинга 16 происходит вычисления позиции указателя мыши в пределах текущего экрана. После этого в строке 11 листинга 16 объявляется объект структуры события и заполняется необходимыми данными (положение курсора в окне и в пределах экрана). После этого идёт рассылка данных об этом событие во все обработчики этого события.

Если пользователь в пределах окна нажал на клавишу мыши, то окно получает уведомление, которое соответствует тому, что именно пользователь нажал или отпустил. В Win32 API для уведомления о нажатии основных клавиш мыши используются девять сообщений: WM\_LBUTTONUP, WM\_RBUTTONUP, WM\_MBUTTONUP, WM\_LBUTTONDOWN, WM\_RBUTTONDOWN, WM\_MBUTTONDOWN, WM\_LBUTTONDBLCLK, WM\_RBUTTONDBLCLK, WM\_MBUTTONDBLCLK. Код обработчика нажатия приведён в листинге 17.

**Листинг 17** – Обработчик события нажатия клавиши мыши

1. **void** Window::m\_WndProcHandler\_MouseButtons(
2. Window\* wnd, **HWND** hWnd, **UINT** msg,
3. **WPARAM** wParam, **LPARAM** lParam)
4. {
5. POINT cursorPoint;
6. GetCursorPos(&cursorPoint);
8. MouseKeyCodes keyCode;
9. MouseKeyClick keyClick;
10. KeyStatus keyStatus;
12. **switch** (msg) {
13. **case** WM\_LBUTTONUP: **case** WM\_RBUTTONUP: **case** WM\_MBUTTONUP: {
14. keyStatus = KEY\_RELEASED;
15. } **break**;
16. **default**: { keyStatus = KEY\_PRESSED; } **break**;
17. }
19. **switch** (msg) {
20. **case** WM\_LBUTTONDOWN: **case** WM\_LBUTTONUP: **case** WM\_LBUTTONDBLCLK: {
21. keyCode = MOUSE\_LEFT;
22. } **break**;
23. **case** WM\_RBUTTONDOWN: **case** WM\_RBUTTONUP: **case** WM\_RBUTTONDBLCLK: {
24. keyCode = MOUSE\_RIGHT;
25. } **break**;
26. **case** WM\_MBUTTONDOWN: **case** WM\_MBUTTONUP: **case** WM\_MBUTTONDBLCLK: {
27. keyCode = MOUSE\_MIDDLE;
28. } **break**;
29. }
31. **switch** (msg) {
32. **case** WM\_LBUTTONDBLCLK: **case** WM\_RBUTTONDBLCLK: **case** WM\_MBUTTONDBLCLK: {
33. keyClick = MOUSE\_CLICK\_DOUBLE;
34. } **break**;
35. **default**: {
36. keyClick = MOUSE\_CLICK\_ONE;
37. } **break**;
38. }
40. MouseEventClick mouseEventClick(
41. keyCode, keyClick, keyStatus,
42. LOWORD(lParam), HIWORD(lParam),
43. cursorPoint.x, cursorPoint.y
44. );
46. **for** (auto handler : wnd->\_mouseClickHandlers) {
47. handler(mouseEventClick);
48. }
49. }

С пятой по десятую строки листинга 17 идёт объявление переменных и определение положения курсора мыши относительно текущего дисплея. Затем, начиная со строки 12 листинга 17 выполняются три ветвления. Первое ветвление определят состояние клавиши (нажата она или отпущена). Второе ветвление определяет, какая именно клавиша была нажата (правая, левая или центральная). Третье ветвление определяет, сколько кликов было сделано (одинарный клик или двойной клик). В 40 строке листинга 17 объявляется структура события и сразу же заполняется данными, после чего эти данные рассылаются в методы-обработчики.

Для определения того, находится ли курсор над окном или нет, в конструкторе (см. листинг 2) был зарегистрирован обработчик события таймера. Каждый раз, когда курсор попадает на элемент управления или покидает его начинается процесс рассылки данной информации в обработчики при условии, что данная возможность включена (см. листинг 1 и комментарии к нему). Код обработчика события входа курсора в область окна и покидание курсором области окна представлен в листинге 18.

**Листинг 18** – Обработчик события наведение курсора

1. **void** Window::m\_WndProcHandler\_MouseHover(
2. Window\* wnd, **HWND** hWnd, **UINT** msg,
3. **WPARAM** wParam, **LPARAM** lParam)
4. {
5. **for** (auto handler : wnd->\_hoverHandlers) {
6. handler(wParam);
7. }
8. }

Каждый раз, когда происходит наведение курсора в область окна, окну отправляется сообщение, в котором параметр wParam имеет значение true или false. В зависимости от этих значений определяется, находится ли курсор в области(true) окна или нет(false).

Когда пользователь вращает колёсиком мыши окну приходит сообщение WM\_MOUSEWHEEL, которое содержит информацию о направление скроллинга. Код обработчика сообщения WM\_MOUSEWHEEL представлен в листинге 19.

**Листинг 19** – Обработчик события WM\_MOUSEWHEEL

1. **void** Window::m\_WndProcHandler\_MouseWheel(
2. Window\* wnd, **HWND** hWnd, **UINT** msg,
3. **WPARAM** wParam, **LPARAM** lParam)
4. {
5. **HWND** hWnd\_pos = WindowFromPoint(POINT{ LOWORD(lParam), HIWORD(lParam) });
6. **if** (!hWnd\_pos) {
7. **return**;
8. }
10. **short** value = GET\_WHEEL\_DELTA\_WPARAM(wParam) / WHEEL\_DELTA;
11. MouseWheelCodes mouseWheelCodes;
12. **if** (value > 0) {
13. mouseWheelCodes = MOUSE\_WHEEL\_TOP;
14. }
15. **else** **if** (value < 0) {
16. mouseWheelCodes = MOUSE\_WHEEL\_BOT;
17. }
19. MouseEventWheel mouseEventWheel(mouseWheelCodes,
20. LOWORD(lParam), HIWORD(lParam),
21. LOWORD(lParam), HIWORD(lParam));
23. **if** (s\_windowsMap.find(hWnd\_pos) != s\_windowsMap.end()) {
24. **for** (auto handler : s\_windowsMap[hWnd\_pos]-> \_mouseWheelHandlers) {
25. handler(mouseEventWheel);
26. }
27. s\_windowsMap[hWnd\_pos]->redrawWindow(**false**);
28. }
29. }

В строке 5 листинга 19 сначала определяется, над каким окном находится курсор в момент, когда был сделано вращение курсора. Если окон найдено не было, то обработка вращения на этом заканчивается. Далее идёт получение информации о вращение колёсика мыши и определяется направление вращения. После этого инициализируется структура с данными о направление вращения и положением курсора, после чего эти данные отправляются окну, которому они адресованы, если оно найдено в списке окон (см. строка 2 листинга 1).

Когда пользователь ставит таймер, окно начинает получать регулярные уведомления о его срабатывание через событие WM\_TIMER. Код обработчика сообщения WM\_TIMER представлен в листинге 20.

**Листинг 20** – Обработчик события WM\_TIMER

1. **void** Window::m\_WndProcHandler\_Timer(
2. Window\* wnd, **HWND** hWnd, **UINT** msg,
3. **WPARAM** wParam, **LPARAM** lParam)
4. {
5. **for** (auto handler : wnd->\_timerHandlers) {
6. handler(wParam);
7. }
8. }

Как только приходит уведомление от таймера, то сразу же в обработчики этого события отправляется идентификатор таймера.

Сообщение WM\_NCCALCSIZE отправляется тогда, когда размер и позиция рабочей области окна должны быть вычислены. Код обработчика сообщения представлен в листинге 21.

**Листинг 21** – Обработчик события WM\_NCCALCSIZE

1. **int** Window::m\_WndProcHandler\_NcCalcSize(
2. Window\* wnd, **HWND** hWnd, **UINT** msg,
3. **WPARAM** wParam, **LPARAM** lParam)
4. {
5. **if** (wParam) {
6. NCCALCSIZE\_PARAMS\* Params = **reinterpret\_cast**<NCCALCSIZE\_PARAMS \*>(lParam);
7. Params->rgrc[0].bottom += wnd->\_borderSize;
8. Params->rgrc[0].right += wnd->\_borderSize;
9. Params->rgrc[1].bottom += wnd->\_borderSize;
10. Params->rgrc[1].right += wnd->\_borderSize;
11. Params->rgrc[2].bottom += wnd->\_borderSize;
12. Params->rgrc[2].right += wnd->\_borderSize;
13. **return** 0;
14. }
15. DefWindowProc(hWnd, msg, wParam, lParam);
16. }

После получения указателя на структуру NCCALCSIZE\_PARAMS идёт увеличение прямоугольников. Первый содержит новые координаты окна, которое было перемещено или изменено, то есть это - предполагаемые новые координаты окна. Второй содержит координаты окна до того, как это было перемещено или изменено. Третий содержит координаты рабочей области окна до того, как окно было перемещено или изменено. Если окно - дочернее окно, координаты отсчитываются относительно рабочей области родительского окна. Делается это для того чтобы избежать дёрганья окна во время изменения его размера.

Сообщение WM\_NCHITTEST отправляется в окно тогда, когда перемещается курсор или когда кнопка мыши нажимается или отпускается. Код обработчика события WM\_NCHITTEST представлен в листинге 22.

**Листинг 22** – Обработчик события WM\_NCHITTEST

1. **int** Window::m\_WndProcHandler\_NcHitTest(
2. Window\* wnd, **HWND** hWnd, **UINT** msg,
3. **WPARAM** wParam, **LPARAM** lParam)
4. {
5. POINT pos;
6. GetCursorPos(&pos);
7. RECT WindowRect;
8. GetWindowRect(hWnd, &WindowRect);
10. **int** x = pos.x - WindowRect.left;
11. **int** y = pos.y - WindowRect.top;
13. **if** (x >= wnd->\_borderSize && x <= wnd->\_width - wnd->\_borderSize
14. && y >= wnd->\_borderSize && y <= MAIN\_WINDOW\_HEADER\_HEIGHT
15. && wnd->\_haveHeader)
16. **return** HTCAPTION;
17. **else** **if** (x <= wnd->\_borderSize && wnd->\_canBeResize\_left)
18. **return** HTLEFT;
19. **else** **if** (y <= wnd->\_borderSize && wnd->\_canBeResize\_top)
20. **return** HTTOP;
21. **else** **if** (x >= wnd->\_width - wnd->\_borderSize - 1 && wnd-> \_canBeResize\_right)
22. **return** HTRIGHT;
23. **else** **if** (y >= wnd->\_hieght - wnd->\_borderSize - 1 && wnd-> \_canBeResize\_bottom)
24. **return** HTBOTTOM;
25. **else**
26. **return** HTCLIENT;
27. }

С пятой строки по одиннадцатую строку листинга 22 вычисляются координаты курсора в окне, после чего идёт определение того, находится ли курсор в пределах границы и, если эта граница активна, посылает операционной системе сигнал о том, что курсор находится на границе и это окно готово менять свой размер.

Чтобы закрыть окно без уничтожения объекта окна необходимо вызывать метод destroy, код которого приведён в листинге 23.

**Листинг 23** – Код метода destroy

1. **void** Window::destroy()
2. {
3. **if** (\_thisWindowIsCreated) {
4. SendMessage(\_hWnd, WM\_CLOSE, 0, 0);
5. \_thisWindowIsCreated = **false**;
6. s\_windowsMap.erase(\_hWnd);
8. **while** (!\_childList.empty()) {
9. \_childList.front()->destroy();
10. }
12. **if** (\_parent) {
13. \_parent->m\_removeChildWindow(**this**);
14. }
15. }
16. }

Данный метод проверяет, создано ли окно, и, если окно создано, посылает ему сообщение о закрытие, после чего устанавливает флаг окна в ложное состояние, чтобы это окно можно было открыть повторно, и удаляет себя из списка окон (см. листинг 1, строка 2). После этого для всех дочерних окон вызывается этот же метод, пока все окна, принадлежавшие этому окну, не будут уничтожены. В конце, если у окна есть родитель, то окно удаляется из списка дочерних окон родителя.

### 3.1.2 Детальная реализация тестового приложения

При реализации тестового приложения было использован материал, описанный в пункте 3.1.1. Некоторые общие компоненты выполняют практически одни и те-же задачи, поэтому часть из них описана не будет.

Построение тестового приложение начата с реализации расширения класса Window – MainWindow. Код прототипа класса представлен в листинге 24.

**Листинг 24** – Прототип класса MainWindow

1. **class** MainWindow : **public** Window {
2. **private**:
3. ButtonClose buttonClose;
4. ButtonMaximize buttonMaximize;
5. ButtonMinimize buttonMinimize;
6. ListOfFiles listOfFiles;
7. ButtonReturn buttonBackward;
8. ButtonReturn buttonForward;
9. ButtonReturn buttonUp;
11. **bool** \_batteryFound;
12. **bool** \_isCharging;
13. **int**  \_batteryStatus;

16. **public**:
17. MainWindow();
19. **virtual** **void** eventCreateWindow() override;
20. **void** paintHandler(Gdiplus::Graphics& graphics);
22. **void** mouseClickHandler(**const** MouseEventClick& mouseEventClick);
23. **void** mouseMoveHandler(MouseEvent& mouseEvent);
24. **void** keyboardHandler(**const** KeyEvent& keyEvent);
26. **void** updateBatteryStatusTimerHandler(**int** timerID);
28. **private**:
30. **void** m\_createChildWindows();
31. **void** m\_registerHandlers();
32. };

В строках 3-9 объявлены поля других реализаций класса Window, которые станут дочерними окнами главного окна. Так же в класс главного окна включены методы-обработчики событий, связанных с движением мыши, кликом мыши, рисованием и нажатием клавиш клавиатуры. Дополнительно вынесены методы создания дочерних окон и регистрация обработчиков.

Окна, которые реагируют на наведение и на клик мыши, расширяют класс Button, который в свою очередь расширяет класс Window. Основная задача кнопок – предоставление интерфейса кнопки для его последующего расширения. Сейчас же они выполняют минимальную задачу. В тестовом приложение реализованы 6 кнопок: кнопка закрытия приложения, кнопка разворачивания окна, кнопка сворачивания окна и три кнопки перехода между каталогами. Далее будут рассмотрены обработчики события нажатия на эти кнопки. В листинге 25 представлен код обработчика кнопки закрытия программы.

**Листинг 25** – Обработчик кликов мыши кнопки завершения программы

1. **void** ButtonClose::mouseClickCloseHandler(
2. **const** MouseEventClick& mouseEventClick)
3. {
4. **if** (getParent()) {
5. getParent()->destroy();
6. }
7. }

Обработчик, представленный в листинге 25, вызывает для своего родителя метод закрытия окна. Родительское окно закрывает все свои дочерние окна и закрывается само.

Кнопка масштабирования окна заставляет главное окно изменить свой размер на максимальный в рамках дисплея. Обработчик этой кнопки представлен в листинге 26.

**Листинг 26** – Обработчик кликов мыши кнопки масштабирования окна

1. **void** ButtonMaximize::maximizeHandler(
2. **const** MouseEventClick& mouseEventClick)
3. {
4. **if** (mouseEventClick.Button == MOUSE\_LEFT
5. && mouseEventClick.Status == KEY\_PRESSED) {
6. **if** (getParent()) {
7. **if** (!\_maximized) {
8. \_oldWidth = getParent()->getWidth();
9. \_oldHieght = getParent()->getHieght();
10. \_oldPosX = getParent()->getPosX();
11. \_oldPosY = getParent()->getPosY();
12. getParent()->resizeWindow(0, 0,
13. GetSystemMetrics(SM\_CXSCREEN),
14. GetSystemMetrics(SM\_CYSCREEN),
15. **true**
16. );
18. \_maximized = **true**;
19. }
20. **else** {
21. getParent()->resizeWindow(
22. \_oldPosX, \_oldPosY,
23. \_oldWidth, \_oldHieght,
24. **true**
25. );
27. \_maximized = **false**;
28. }
29. }
30. }
31. }

Кнопка масштабирования окна хранит текущий статус масштабирования. Если окно не масштабировано, то оно увеличивается в своих размерах, иначе оно возвращается в состояние, установленное перед масштабированием.

Кнопка сворачивания окна прячет окно. На листинге 27 представлен код, скрывающий окно приложения.

**Листинг 27** – Обработчик кликов мыши кнопки сворачивания окна

1. **void** ButtonMinimize::minimizeHandler(
2. **const** MouseEventClick& mouseEventClick)
3. {
4. **if** (getParent()) {
5. getParent()->minimizeWindow(**true**);
6. }
7. }

В пятой строке листинга 27 для того чтобы свернуть окно вызывается метод сворачивания окна.

Последние три кнопки отвечают за управление списком файлов. Первая кнопка отвечает за перемещение на один каталог выше, то есть перемещение «назад». Вторая кнопка отвечает за возвращение в каталог, из которого было совершено перемещение в текущий. Последняя кнопка отвечает за обратное действие для второй кнопки. Обработчики этих кнопок находятся в классе, реализующем список файлов. Коды обработчиков приведены в листинге 28.

**Листинг 28** – Обработчики кликов управления списком файлов

1. **void** ListOfFiles::mouseClickButtonBackwardHandler(
2. **const** MouseEventClick& mouseEventClick)
3. {
4. **if** (mouseEventClick.Click == MOUSE\_CLICK\_ONE
5. && mouseEventClick.Button == MOUSE\_LEFT
6. && mouseEventClick.Status == KEY\_PRESSED
7. && !\_buttonBackward->isLocked()) {
8. **if** (!\_backwardStack.empty()) {
9. \_forwardStack.push(\_thisDirection);
10. \_thisDirection = \_backwardStack.top();
11. \_backwardStack.pop();
12. **if** (\_backwardStack.empty()) {
13. \_buttonBackward->setLock(**true**);
14. \_buttonBackward->redrawWindow(**false**);
15. }
17. redrawWindow(**false**);
18. \_buttonForward->setLock(**false**);
19. \_buttonForward->redrawWindow(**false**);
21. updateList();
22. }
23. }
24. }
25. **void** ListOfFiles::mouseClickButtonForwardHandler(
26. **const** MouseEventClick& mouseEventClick)
27. {
28. **if** (mouseEventClick.Click == MOUSE\_CLICK\_ONE
29. && mouseEventClick.Button == MOUSE\_LEFT
30. && mouseEventClick.Status == KEY\_PRESSED
31. && !\_buttonForward->isLocked()) {
32. **if** (!\_forwardStack.empty()) {
33. \_backwardStack.push(\_thisDirection);
34. \_thisDirection = \_forwardStack.top();
35. \_forwardStack.pop();
37. **if** (\_forwardStack.empty()) {
38. \_buttonForward->setLock(**true**);
39. \_buttonForward->redrawWindow(**false**);
40. }
42. redrawWindow(**false**);
43. \_buttonBackward->setLock(**false**);
44. \_buttonBackward->redrawWindow(**false**);
46. updateList();
47. }
48. }
49. }

52. **void** ListOfFiles::mouseClickButtonUpHandler(
53. **const** MouseEventClick& mouseEventClick)
54. {
55. **if** (mouseEventClick.Click == MOUSE\_CLICK\_ONE
56. && mouseEventClick.Button == MOUSE\_LEFT
57. && mouseEventClick.Status == KEY\_PRESSED
58. && !\_buttonUp->isLocked()) {
59. std::wstring currentDirection = \_thisDirection;
61. **if** (!currentDirection.empty()) {
62. File file(currentDirection);
63. std::wstring backPath = file.getPrevDirection();
64. nextDirrectory(\_thisDirection);
65. \_thisDirection = backPath;
66. updateList();
68. **if** (backPath.empty()) {
69. setLock(**true**);
70. }
71. **else** {
72. setLock(**false**);
73. }
75. updateList();
76. }
77. }
78. }

## 3.2 Сопроводительная документация

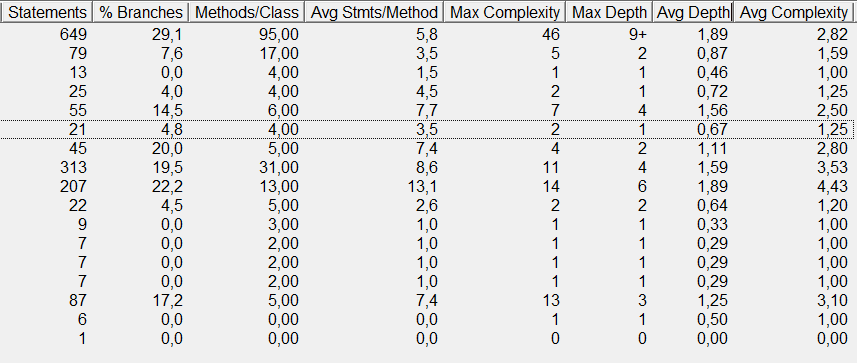
Сопроводительная документация по разработанному программному продукту представлена техническим заданием (приложение А) согласно ГОСТ 19.201-78. Требования к сопроводительной документации устанавливаются государственными стандартами ЕСПД

## 3.3 Анализ ПО

С помощью бесплатной программы SourceMonitor были проанализированы метрики кода (без учёта заголовочных файлов). Результат анализа представлен на рисунке 3.4.1.

Показатели удовлетворительные, за исключением показателей класса Window – у этого класса показатель «Max Complexity» имеет значение 46, что было бы плохо, если бы в нём не было метода, распределяющего сообщения между окнами. Так же этот класс имеет плохой показатель максимальный глубины, но тот факт, что в нём находится метод, распределяющий сообщения, оправдывает столь низкие показатели.

Общее количество строк кода составляет около 2000 строк, без учёта пустых строк. Без учёта заголовочных файлов выходит 1500 строк кода. Самым большим классом, как и ожидалось, стал класс Window.

Рисунок 3.4.1 – метрики кода программы.

## 3.4 Тестирование ПО

Для тестирования класса Window, необходимо провести smoke test.

Список тестов, которые необходимо провести:

1. Создание окна.
2. Перемещение окна.
3. Изменение размеров окна.
4. Рисование в окне.
5. Создание дочернего окна.
6. Регистрация обработки события дочернего окна.
7. Регистрация обработки события родительского окна.
8. Завершение приложения при закрытие всех окон.

Для создания окна необходимо создать объект окна, после чего вызывать метод создания окна. В результате должно быть получена полностью пустое окно. Реальный результат выполнения этих действий представлен на рисунке 3.4.1.

Большой чёрный прямоугольник на рисунке 3.4.1 является пустым окном. Так же в меню многозадачности появился процесс этого окна, что означает, что данный тест пройден.

Активируем окну заголовок, чтобы его можно было перетащить. Результат переноса окна показан на рисунке 3.4.2.

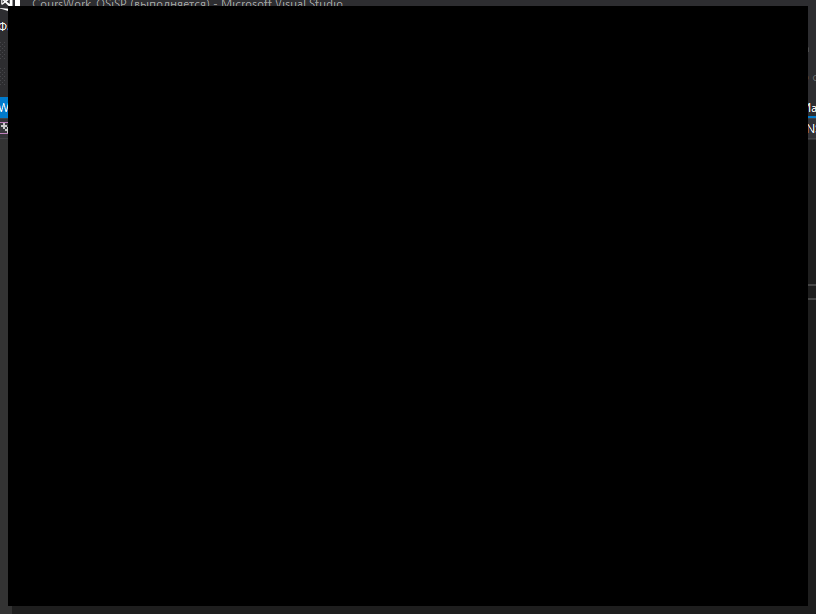


Рисунок 3.4.1 – результат создания окна.

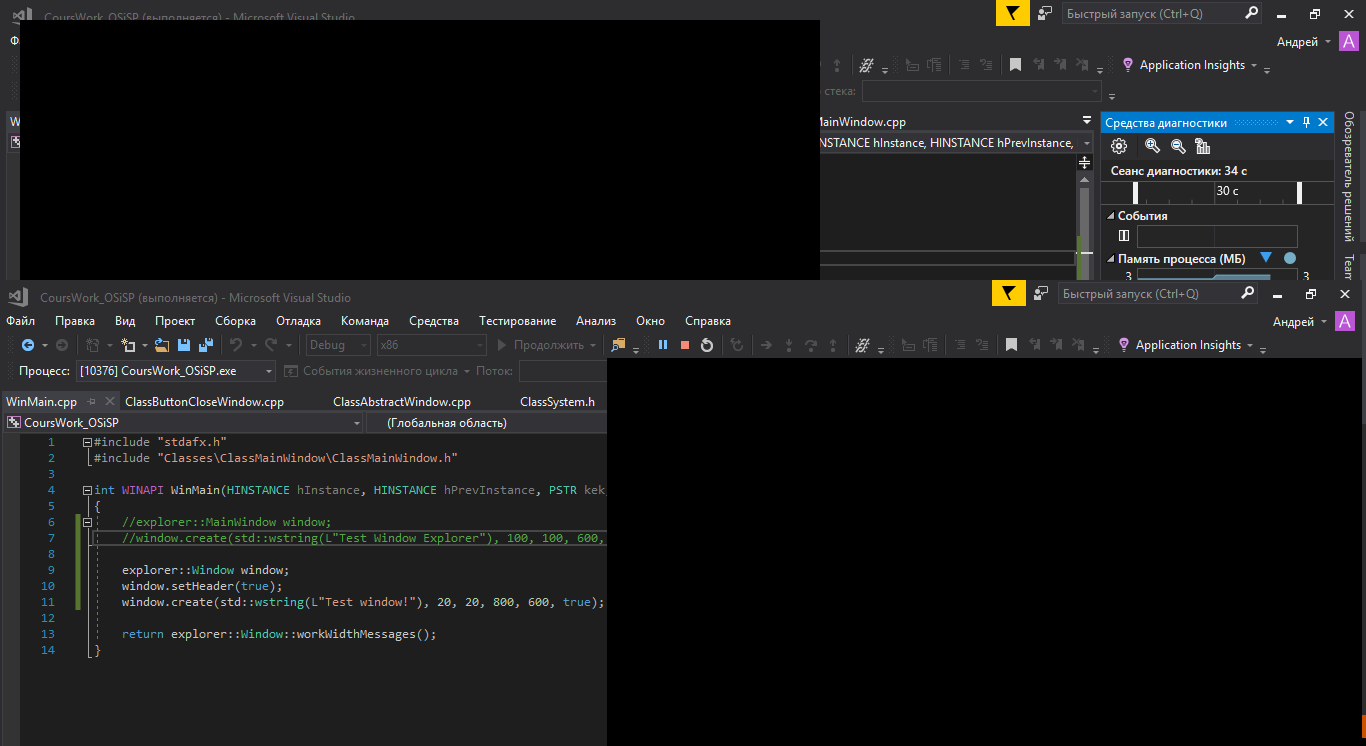


Рисунок 3.4.2 – результат перемещения окна.

На рисунке 3.4.2 совмещены два рисунка. Вверху видно одно положение чёрного окна, внизу видно новое положение этого чёрного окна. Это означает, что окно было перемещено, а тест – пройден.

Добавим окну границы и изменим его размер. Результат изменения размера показан на рисунке 3.4.3.

На рисунке 3.4.3 видно, что окно в левой части рисунка меньше, чем окно в правой части окна. Так же видно, что при создании они получили одинаковые размеры, что означает, что тест пройден.

Создадим обработчик рисования, чтобы проверить, будет ли рисоваться тестовое изображение в окне. Результат приведён на рисунке 3.4.4.

Результат рисования в окне виден на рисунке 3.4.4, следовательно, тест успешно пройден.

Создадим ещё один объект окна и сделаем его дочерним к главному окну. Для наглядности нарисуем в дочернем окне круг. Результат теста приведён на рисунке 3.4.5.

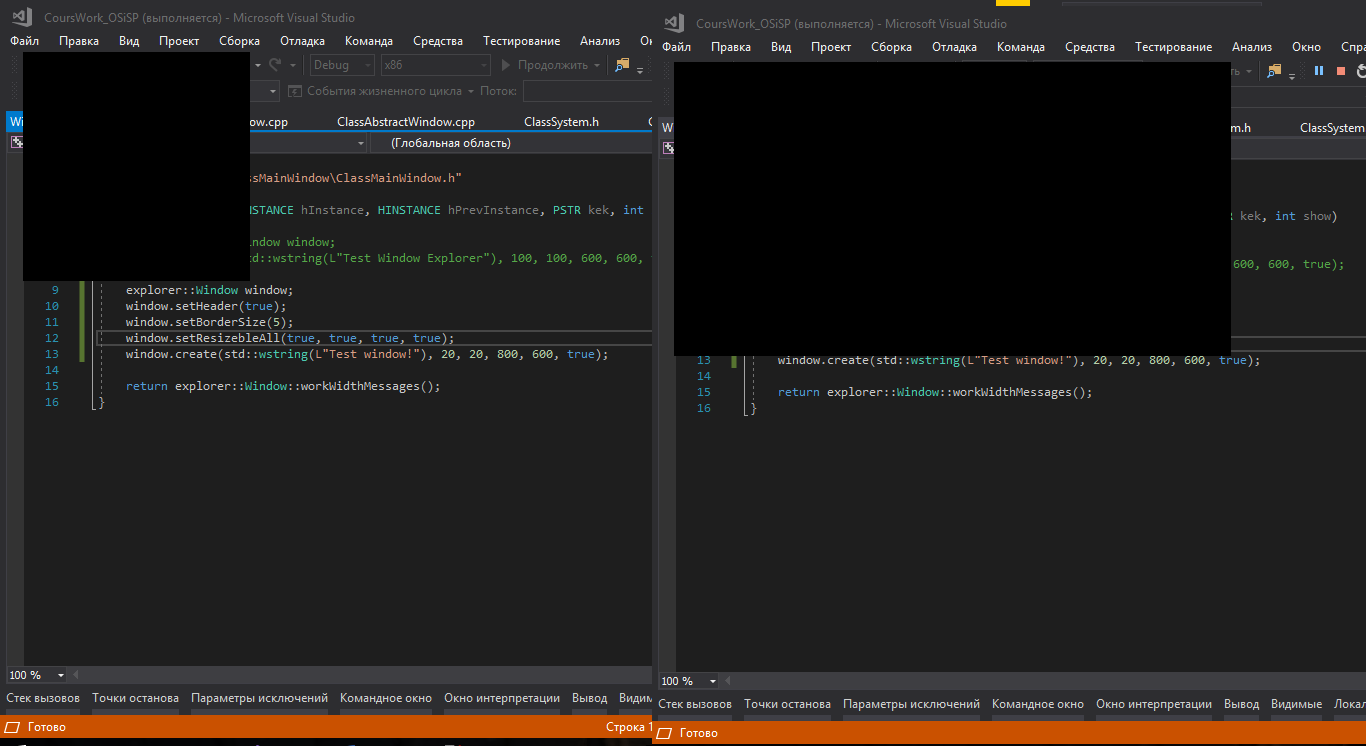


Рисунок 3.4.3 – результат изменения размеров окна.

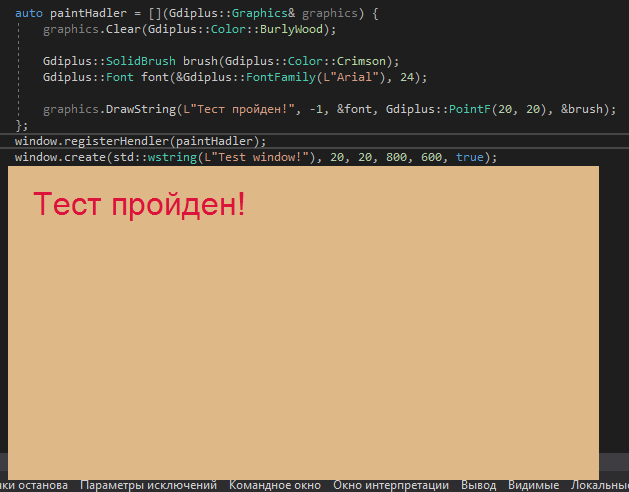


Рисунок 3.4.4 – результат рисования в окне.

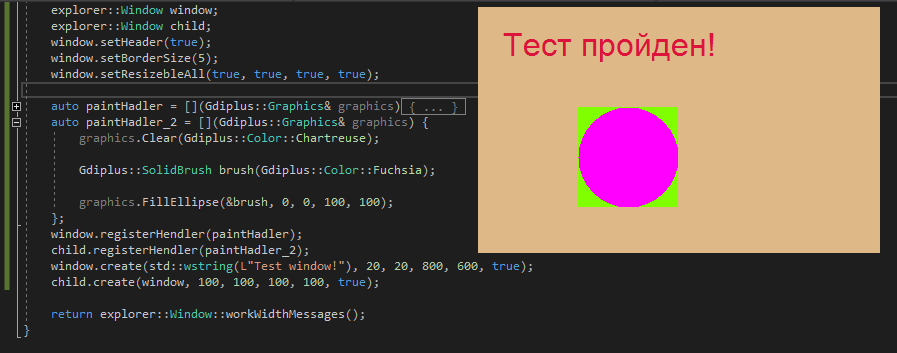


Рисунок 3.4.5 – результат создания дочернего окна.

В главном окне появилось окно с нарисованным кругом, следовательно, тест пройден.

Последние два теста будут проведены вместе. При нажатии на дочернее окно будет изменяться цвет текста в родительском, а при нажатии на родительское окно будет изменяться цвет круга в дочернем. Результат теста приведён на рисунке 3.4.6.



Рисунок 3.4.6 – результат взаимной подписки на событие клика мыши.

На рисунке 3.4.6 видно, что цвета сфер и текста изменились в зависимости от того, куда был сделан клик – тест пройден.

Из проведённых тестов следует, что данный класс выполняет свой задачи по формированию интерфейса пользователя.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проектирования объектно-ориентированной обёртки я столкнулся с рядом трудностей, из которых одна связана с одной из используемых библиотек – GDI+. Основная проблема этой библиотеки – её медлительность. В будущем необходимо будет выполнить переход на более высокоуровневую технологию – Direct2D или DirectX11/12.

Ещё более сложной задачей было спроектировать данную объектно-ориентированную «обёртку», так как важно было учесть множество моментов, связанных с обработкой различных оконных сообщений. Получившаяся архитектура класса окна не претендует на идеальную и требует серьёзных доработок, однако проделанная работа уже является удовлетворительной, так как теперь для создания окна в классическом приложение не надо писать много кода, как это было в случае с «чистым» Win32 API. Если привести существующие реализацию в полной порядок и выполнить оптимизации и улучшения, то можно добиться максимальной простоты создания и использования ОО «обёртки».

Так же в процессе выполнения проектирования были получены новые знания, связанные использованием нового API, появившегося в операционной системе Windows начиная с версии 6.0 – DWM API [14]. Эта технология является современным решением для построения составных окон и даже была использована в тестовом приложении. Дальнейшее движение в эту сторону позволит разработать интерфейс, упрощающий создание пользовательского интерфейса с привлекательным внешним видом.

Несмотря на трудности мной был спроектирован интерфейс класса окна, предоставляющий функционал, который упрощает процесс манипуляцией окном. Программисту больше не надо создавать глобальные переменные или создавать кучу обработчиков на все случаи жизни. Теперь программисту достаточно расширить класс окна своим функционалом и создать один объект этого класса, в котором будет хранится информация только об этом объекте, и будут выполнятся те действия, которые будут задуманы программистом.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Язык программирования Java / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/Java>. Дата обращения: 06.09.2017.
2. Язык программирования C# / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/C_Sharp>. Дата обращения: 06.09.2017.
3. Язык программирования C / Википедия [Электронный ресурс] –– Режим доступа: https://ru.wikipedia.org. Дата обращения: 06.09.2017.
4. Язык программирования C++ / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/C%2B%2B>. Дата обращения: 06.09.2017.
5. Microsoft Foundation Classes / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/Microsoft_Foundation_Classes>. Дата обращения: 08.09.2017.
6. Windows Template Library (WTL) / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/Windows_Template_Library>. Дата обращения: 08.09.2017.
7. QT / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/Qt>. Дата обращения: 08.09.2017.
8. GIMP ToolKit (GTK+) / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/GTK%2B>. Дата обращения: 08.09.2017.
9. wxWidgets / Википедия [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/WxWidgets>. Дата обращения: 08.09.2017.
10. Graphics Device Interface Plus / MSDN [Электронный ресурс] –– Режим доступа: <https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms533798(v=vs.85).aspx>. Дата обращения: 08.09.2017.
11. Direct2D / MSDN [Электронный ресурс] –– Режим доступа: <https://msdn.microsoft.com/ru-ru/library/windows/desktop/dd370990(v=vs.85)>. Дата обращения: 08.09.2017.
12. DirectX / MSDN [Электронный ресурс] –– Режим доступа: <https://ru.wikipedia.org/wiki/DirectX>. Дата обращения: 08.09.2017.
13. cppreference.com [Электронный ресурс] –– Режим доступа: <http://ru.cppreference.com/w/> . Дата обращения: 04.10.2017.
14. Desktop Window Manager / MSDN [Электронный ресурс] –– Режим доступа: <https://msdn.microsoft.com/en-us/library/windows/desktop/aa969540(v=vs.85).aspx>. Дата обращения: 25.11.2017.
15. Microsoft Developer Network / MSDN [Электронный ресурс] –– Режим доступа: [https://msdn.microsoft.com](https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx) . Дата обращения: 11.10.2017.

ПРИЛОЖЕНИЕ А

(обязательное)

**Техническое задание**

**Введение**

ОО «обёртка» над Win32 API предназначена для упрощения и ускорения разработки программного обеспечения на языке C++.

**А.1 Основание для разработки**

Программный продукт разрабатывается в рамках курсового проекта студента учреждения образования «Полоцкий государственный университет» Стеняева А. Д. Основанием для разработки является отсутствие совместимости с низкоуровневым Win32 API у большинства популярных библиотек.

**А.2 Назначение разработки**

Назначение программного продукта – ускорение разработки программного обеспечения за счёт организации объектно-ориентированного подхода к разработке. Предоставление высокоуровнего интерфейса для работы с низкоуровневым функционалом, с возможностью использования низкоуровневой составляющей.

**А.3 Требования к программе или программному изделию**

**А.3.1 Требования к функциональным характеристикам**

При разработке программного продукта были выдвинуты следующие требования к функциональным характеристикам:

1. Программное средство должно предоставлять высокоуровневый интерфейс для создания приложений для операционной системы Windows.
2. Совместимость с низкоуровневым API.
3. Использование продукта должно быть максимально простым.

Разработанное приложение должно иметь следующие элементы и функций:

1. Создание окна.
2. Изменение размеров окна.
3. Изменение положения окна.
4. Рисование в окне.
5. Регистрация обработчиков событий.
6. Закрытие окна.
7. Получение системной информации.

Ограничения по временным характеристикам функционирования системы отсутствуют.

**А.3.2 Требования к надежности**

Программный продукт должен показывать высокую стабильность работы и быть устойчивым к сбоям.

**А.3.3 Условия эксплуатации**

Для эксплуатации программы ОО «обёртки» необходимо понимание принципов объектно-ориентированного подхода к программированию, а также понимание структуры приложений, разрабатываемых для операционной системы Windows.

**А.3.4 Требования к составу и параметрам технических средств**

Для обеспечения устойчивости работы программного средства требуется:

1. Процессор Intel Pentium 4 и выше
2. ОЗУ не менее 1 ГБ (рекомендуется 4 ГБ).
3. До 32 ГБ доступного дискового пространства.

**А.3.5 Требования к информационной и программной   
совместимости**

1. ОС Windows Vista и выше.
2. Наличие клавиатуры.
3. Наличие мыши.
4. Наличие монитора.

**А.3.6 Требования к маркировке и упаковке**

Требования к маркировке и упаковке отсутствуют.

**А.3.7 Требования к транспортированию и хранению**

Программное средство должно храниться на электронном носителе в виде исходных файлов и в виде библиотек.

**А.4 Требования к программной документации**

Программная документация по управление приложением «Windows explorer» должна быть предоставлена в следующем составе технического задания. Согласно ГОСТ 19.201-78.

Требования к перечисленным программным документам устанавливаются государственными стандартами ЕСПД.

**А.5 Стадии и этапы разработки**

Разработка программного средства «Windows explorer» должна происходить в следующем порядке:

1. Анализ исходных данных и постановка задачи проектирования, разработка технического задания.
2. Проектирование.
3. Разработка интерфейса, архитектуры и структуры программных модулей.
4. Реализация и тестирование программы.
5. Разработка программной документации.

**А.6 Порядок контроля и приемки**

Контроль и приемка программного средства осуществляется в соответствии с программой и методикой испытаний.

Для проверки корректности приложения применялись следующие программные средства:

1. ОС Windows 10 PRO x64.
2. Среда разработки Visual Studio 2017 Community Edition.

Тестирование программы состояло из проверки корректности работы ранее перечисленных функций.

Методы испытаний:

Основным методом испытания программы является визуальный контроль выполнения программой требующихся функций, корректное выполнение smoke тестов.

ПРИЛОЖЕНИЕ Б

(обязательное)

Диаграмма вариантов использования

