# Состав модулей и интерфейса программы Архиватор

Программа состоит из 3 основных модулей:

1. Главный модуль (**source.py**)

Является точкой входа в программу, запускает основное окно проводника с выбором файлов.

2. Модуль проводника (**FileExplorer.py**)

Представляет из себя графический интерфейс программы, отвечает за работу окон, кнопок и полей ввода, а также управляет основными потоками сжатия и распаковки.

3. Модуль архиватора (**ArchieveTreads.py**)

Описывает работу потоков сжатия и распаковки архивов, благодаря чему пользователь может приостановить процесс без закрытия программы.

*Листинг 1 source.py*

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QTreeView, QFileSystemModel, QPushButton, QFileDialog, QLineEdit, QLabel, QProgressDialog

from PyQt5.QtCore import Qt, QThread, pyqtSignal

from PyQt5.QtGui import QIcon

import zipfile

import os

from ArchiveTreads import \*

from FileExplorer import \*

if \_\_name\_\_ == '\_\_main\_\_':

    app = QApplication(sys.argv)

    window = FileExplorer()

    window.show()

    sys.exit(app.exec\_())

*Листинг 2 FileExplorer.py*

import sys

from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QTreeView, QFileSystemModel, QPushButton, QFileDialog, QLineEdit, QLabel, QProgressDialog

from PyQt5.QtCore import Qt, QThread, pyqtSignal

from PyQt5.QtGui import QIcon

import zipfile

import os

from ArchiveTreads import \*

class FileExplorer(QWidget):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.initUI()

    def initUI(self):

        self.setWindowTitle('Архиватор')

        self.setWindowIcon(QIcon('icons/icon.png'))

        self.setGeometry(100, 100, 800, 600)

        # Создаем модель файловой системы

        self.model = QFileSystemModel()

        self.model.setRootPath('')

        # Создаем виджеты представления файловой системы и устанавливаем модель

        self.tree = QTreeView()

        self.tree.setModel(self.model)

        self.tree.setRootIndex(self.model.index(''))  # Устанавливаем корневой индекс

        self.tree.doubleClicked.connect(self.open\_file)

        self.tree.setSelectionMode(QTreeView.ExtendedSelection)  # Разрешаем множественный выбор

        self.tree.setColumnWidth(0, 200)

        self.tree.setSortingEnabled(True)

        # Создаем окно пути

        self.path\_line\_edit = QLineEdit()

        self.path\_line\_edit.setPlaceholderText('Путь к файлам...')

        self.go\_button = QPushButton()

        self.go\_button.setIcon(QIcon("icons/go\_icon.png"))

        self.go\_button.clicked.connect(self.change\_directory)

        self.up\_button = QPushButton()

        self.up\_button.setIcon(QIcon("icons/up\_icon.png"))

        self.up\_button.clicked.connect(self.go\_up)

        #архивацией и поиском

        self.add\_button = QPushButton('Добавить')

        self.extract\_button = QPushButton('Извлечь')

        self.home\_button = QPushButton()

        self.home\_button.setIcon(QIcon("icons/home\_icon.png"))

        # Соединяем сигнал кнопки с методом

        self.home\_button.clicked.connect(self.go\_home)

        self.pc\_button = QPushButton()

        self.pc\_button.setIcon(QIcon("icons/pc\_icon.png"))

        # Соединяем сигнал кнопки с методом

        self.pc\_button.clicked.connect(self.pc)

        # Соединяем сигналы кнопок с методами

        self.add\_button.clicked.connect(self.create\_zip)

        self.extract\_button.clicked.connect(self.extract\_zip)

        # Создаем горизонтальный макет для кнопок управления

        button\_layout = QHBoxLayout()

        button\_layout.addWidget(self.home\_button)

        button\_layout.addWidget(self.pc\_button)

        button\_layout.addWidget(self.up\_button)

        button\_layout.addWidget(self.path\_line\_edit)

        button\_layout.addWidget(self.go\_button)

        button\_layout.addWidget(self.add\_button)

        button\_layout.addWidget(self.extract\_button)

        button\_layout.addWidget(self.path\_line\_edit)

        # Создаем вертикальный макет и добавляем виджет представления файловой системы и кнопки управления

        layout = QVBoxLayout()

        layout.addLayout(button\_layout)

        layout.addWidget(self.tree)

        # Устанавливаем макет для главного окна

        self.setLayout(layout)

        # Подключаем обработчик события выбора элемента в дереве

        self.tree.selectionModel().selectionChanged.connect(self.update\_path\_label)

    def go\_home(self):

        home\_path = os.path.expanduser("~")  # Получаем путь к домашней директории пользователя

        self.model.setRootPath(home\_path)  # Устанавливаем новый корневой путь для модели

        self.tree.setRootIndex(self.model.index(home\_path))  # Обновляем представление

        self.path\_line\_edit.setText(home\_path)  # Обновляем поле ввода пути

    def pc(self):

        self.model.setRootPath('')  # Устанавливаем новый корневой путь для модели

        self.tree.setRootIndex(self.model.index(''))  # Обновляем представление

        self.path\_line\_edit.setText('')  # Обновляем поле ввода пути

    def go\_up(self):

        current\_path = self.path\_line\_edit.placeholderText().strip()  # Получаем текущий путь из плейсхолдера поля ввода

        parent\_path = os.path.dirname(current\_path)  # Получаем родительскую директорию текущего пути

        if os.path.isdir(parent\_path):  # Проверяем, существует ли родительская директория

            self.model.setRootPath(parent\_path)  # Устанавливаем родительскую директорию как новый корневой путь для модели

            self.tree.setRootIndex(self.model.index(parent\_path))  # Обновляем представление

            self.path\_line\_edit.setPlaceholderText(parent\_path)  # Обновляем поле ввода пути

        else:

            self.go\_home()

    def open\_file(self, index):

        # Получаем путь к файлу или директории по индексу

        path = self.model.filePath(index)

        if os.path.isfile(path):

            os.startfile(path)  # Запускаем файл

        elif os.path.isdir(path):

            self.model.setRootPath(path)

            self.tree.setRootIndex(index)

            self.path\_line\_edit.setPlaceholderText(path)

            self.path\_line\_edit.clear()

    def update\_path\_label(self):

        # Получаем путь выбранного элемента и обновляем текст окна пути

        selected\_index = self.tree.currentIndex()

        path = self.model.filePath(selected\_index)

        self.path\_line\_edit.setPlaceholderText(f'{path}')

    def change\_directory(self):

        # Получаем путь из поля ввода

        path = self.path\_line\_edit.text().strip()

        if not path:

            return

        # Устанавливаем новый корневой путь для модели

        self.model.setRootPath(path)

        self.tree.setRootIndex(self.model.index(path))

    def create\_zip(self):

        # Получаем список выбранных файлов и директорий

        selected\_indexes = list(set(self.tree.selectionModel().selectedIndexes()))

        paths = list(set([self.model.filePath(index) for index in selected\_indexes]))

        # Открываем встроенное диалоговое окно для выбора места сохранения архива

        save\_path, \_ = QFileDialog.getSaveFileName(self, 'Сохранить как', '', 'ZIP Archive (\*.zip)')

        if not save\_path:

            return  # Отменено пользователем или не выбран путь сохранения

        # Создаем диалоговое окно процесса сжатия

        progress\_dialog = QProgressDialog(self)

        progress\_dialog.setWindowTitle('Создание архива')

        progress\_dialog.setLabelText('Идет процесс создания архива...')

        progress\_dialog.setCancelButton(None)

        progress\_dialog.setRange(0, 0)  # Бесконечный режим прогресса

        # Запускаем поток сжатия

        compression\_thread = CompressionThread(paths, save\_path)

        compression\_thread.updateProgress.connect(progress\_dialog.setValue)

        compression\_thread.compressionFinished.connect(progress\_dialog.cancel)

        compression\_thread.start()

        # Отображаем диалоговое окно процесса сжатия

        progress\_dialog.exec\_()

    def extract\_zip(self):

        selected\_indexes = list(set(self.tree.selectionModel().selectedIndexes()))

        zips\_path = list(set([self.model.filePath(index) for index in selected\_indexes]))

        # Открываем диалоговое окно для выбора архива для извлечения

        extract\_path = QFileDialog.getExistingDirectory(self, 'Выбрать место извлечения')

        if not extract\_path:

            return  # Отменено пользователем или не выбрано место извлечения

        for zips in zips\_path:

            if not zips:

                return  # Отменено пользователем или не выбран архив

            # Извлекаем архив

            with zipfile.ZipFile(zips, 'r') as zipf:

                zipf.extractall(extract\_path)

*Листинг 3 ArchieveTreads.py*

import os

import zipfile

from PyQt5.QtCore import QThread, pyqtSignal

class CompressionThread(QThread):

    # Сигналы для обновления прогресса и завершения операции

    updateProgress = pyqtSignal(int)

    compressionFinished = pyqtSignal()

    def \_\_init\_\_(self, file\_paths, save\_path, parent=None):

        super().\_\_init\_\_(parent)

        self.file\_paths = file\_paths

        self.save\_path = save\_path

    def run(self):

        # Создаем архив

        with zipfile.ZipFile(self.save\_path, 'w') as zipf:

            for i, path in enumerate(self.file\_paths):

                if os.path.isfile(path):

                    zipf.write(path, os.path.basename(path))  # Добавляем файл в архив

                elif os.path.isdir(path):

                    for root, dirs, files in os.walk(path):

                        for file in files:

                            file\_path = os.path.join(root, file)

                            zipf.write(file\_path, os.path.relpath(file\_path, path))  # Добавляем файлы из директории в архив

                # Отправляем сигнал для обновления прогресса

                self.updateProgress.emit((i + 1) \* 100 // len(self.file\_paths))

        # Отправляем сигнал о завершении операции

        self.compressionFinished.emit()

class UnzipThread(QThread):

    progressChanged = pyqtSignal(int)

    def \_\_init\_\_(self, archive\_path, extract\_to):

        super().\_\_init\_\_()

        self.archive\_path = archive\_path

        self.extract\_to = extract\_to

    def run(self):

        with zipfile.ZipFile(self.archive\_path, 'r') as zip\_ref:

            file\_count = len(zip\_ref.infolist())

            extracted\_files = 0

            for file\_info in zip\_ref.infolist():

                zip\_ref.extract(file\_info, self.extract\_to)

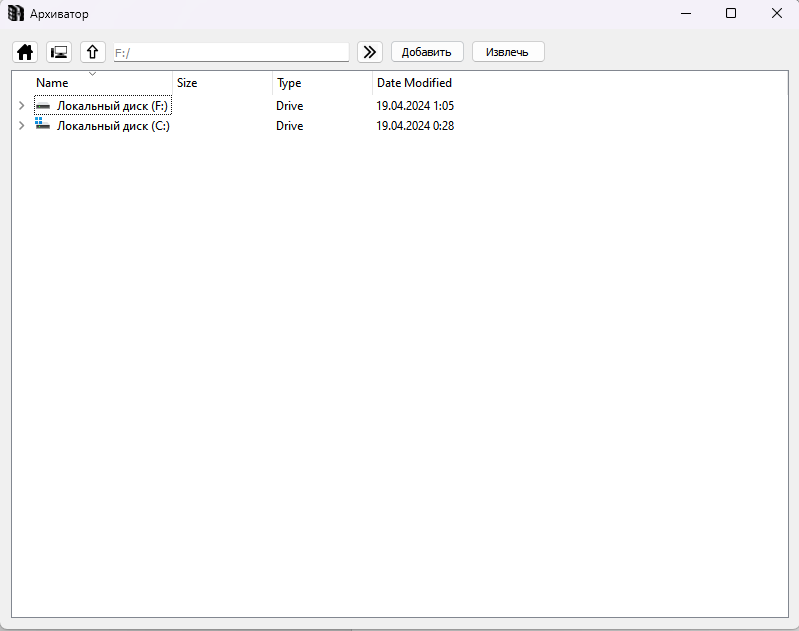
                extracted\_files += 1

                progress = int(extracted\_files / file\_count \* 100)

                self.progressChanged.emit(progress)

# Проверка работоспособности программы

Графический интерфейс программы представляет из себя окно выбора файлов, в котором следует выбрать файлы для архивации или архивы для распаковки:

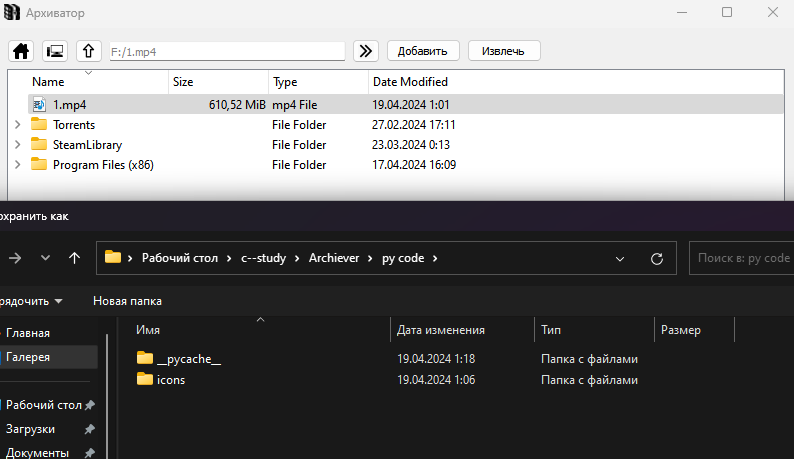


На верхней панели расположены кнопки возврата в директорию пользователя (иконка дома), директорию всех накопителей (иконка компьютера), выхода на путь выше директории (иконка стрелки вверх).

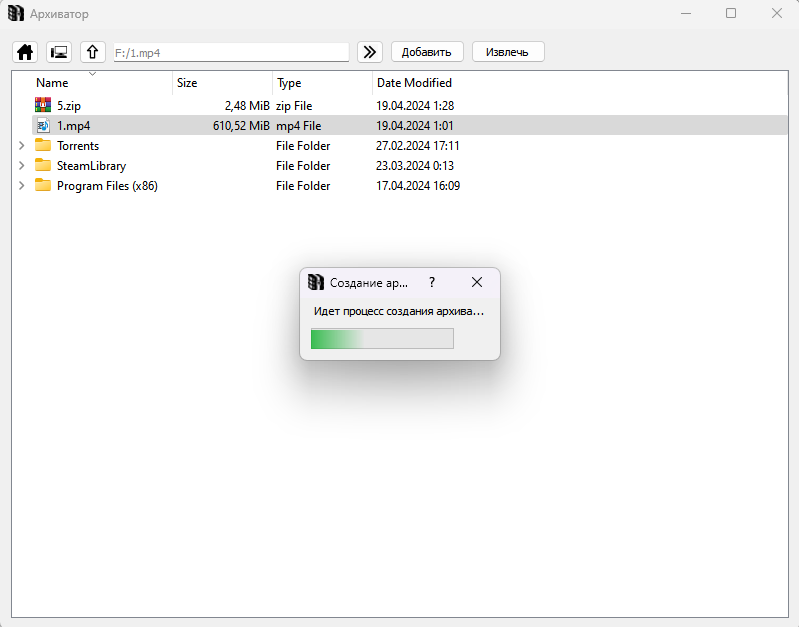
Также там находится текстовое окно текущего пути, в которое можно вписать собственную директорию, к которой необходимо перейти. Для перехода к вписанной директории необходимо нажать на кнопку (иконка двойной стрелки вправо).

Далее расположены кнопки Добавить для добавления выделенных файлов в архив и Извлечь для извлечения выбранных архивов.

При добавлении файла в архив откроется окно выбора расположения и ввода названия архива:

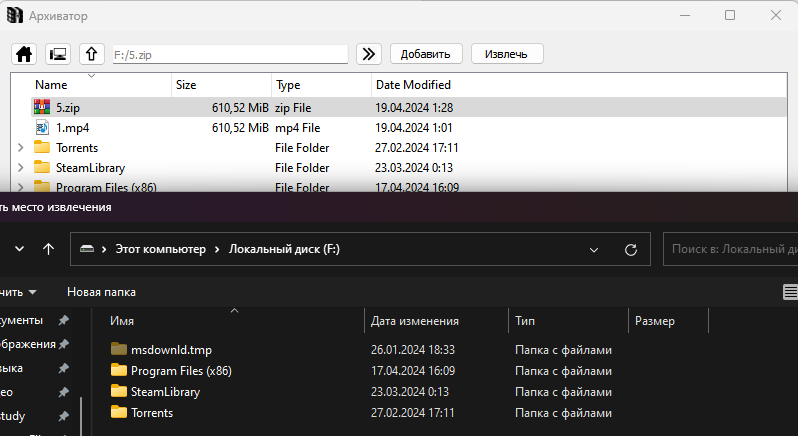


После выбора расположения и ввода имени архива запустится окно прогресса создания архива:



Процесс можно отменить, закрыв окно прогресса, программа будет способна к дальнейшей работе.

При распаковке выбранного архива пользователю также будет предложено выбрать путь распаковки:



Также любые файлы можно запускать из проводника или же заходить в папку дважды нажав по ним.