# **3** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе будут подробно рассмотрены все аспекты дальнейшего функционирования программы. Для этого будет проведён анализ основных модулей, из которых состоит программа и рассмотрим их зависимости. Будут подробно рассмотрены классы и их методы, функции, списки констант и основные механизмы взаимодействия между модулями программы.

В программе можно условно выделить десять частей:

1. Модуль интерфейса пользователя.
2. Модуль описания команд.
3. Модуль журналирования.
4. Модуль настройки приложения.
5. Модуль хранения данных.
6. Модуль анализа файлов.
7. Модуль пересылки сообщений.
8. Модуль управления правилами.
9. Модуль взаимодействия с монитором.
10. Модуль мониторинга файловой системы.

Все модули данного приложения оформлены в виде пакетов языка Python. Классы, константы и функции внутри каждого из этих модулей можно поделить на сущности, обеспечивающие работу данного модуля и сущности, позволяющие другим модулям с ним взаимодействовать. Всё что можно делать с каждым модулем чётко определено интерфейсом этого модуля и\или протоколами взаимодействия. Все интерфейсы и протоколы будут так же описаны в данном разделе.

## **3.1** Модуль интерфейса пользователя

Данный модуль предназначен для обеспечения взаимодействия пользователя с основным приложением. Главной же задачей интерфейса пользователя с программной точки зрения является получение команды от пользователя, преобразование её в строку JSON и отправка её через сокет ZeroMQ модулю описания команд. Такое разделение позволяет реализовывать интерфейс пользователя на разных операционных системах с помощью наиболее подходящего для этих целей языка программирования.

В рамках данного дипломного проекта реализован консольный интерфейс пользователя. Реализован он в модуле terminal\_client. Функции, отвечающие за его реализацию:

* \_parse\_args – функция, которая не принимает ничего и возвращающая словарь с полученными от пользователя аргументами и флагами;
* \_get\_command\_json\_string – функция, которая принимает на вход словарь с аргументами и возвращающая команду, сериализованную в строку JSON;
* main – функция, которая последовательно вызывает две предыдущие команды, и отправляет через интерфейс обмена сообщениями запрос и получает результат в виде строки JSON;

Вторая часть интерфейса пользователя реализована в модуле connector. Он является связующим звеном для пользовательского интерфейса и приложения и служит для приёма команды в виде строки JSON, её десериализации и непосредственной передачи модулю описания команд. К пользовательскому интерфейсу он относится по той причине, что скрывает сам факт его наличия. Внеся незначительные изменения в этот модуль, можно убрать зависимость от пользовательского интерфейса совсем (например, читая настройки и начальные команды из загрузочного файла). Данный модуль запускается как отдельное приложение и имеет следующую функциональность:

* main – функция, которая в бесконечном цикле ожидает приходящие от пользователя команды, и, когда они пришли, отправляет их на обработку. Если есть результат обработки – отправляет его пользователю;
* \_process\_incoming\_command – данная функция принимает сериализованную в строку JSON команду и ссылку на класс описания команд, десериализует её и отправляет на исполнение, возвращает сериализованный в строку JSON результат выполнения команды (если он есть);

## **3.2** Модуль описания команд

Данный модуль предназначен для выполнения команд, поступивших от пользователя. Интерфейс этого модуля достаточно прост:

class CommandsDescriptionInterface(metaclass=ABCMeta):

@abstractmethod

def execute\_command(self, command):

pass

В функцию execute\_command передаётся объект команды, который выглядит определён с помощью класса Command:

class Command:

def \_\_init\_\_(self, target\_block, action, additional\_information):

self.\_target\_block = target\_block

self.\_action = action

self.\_additional\_information = additional\_information

@property

def target\_block(self):

return self.\_target\_block

@property

def action(self):

return self.\_action

@property

def additional\_information(self):

return self.\_additional\_information

Класс команды имеет следующие свойства:

* target\_block – идентификатор целевого блока, к интерфейсу которого нужно обратиться для того что бы выполнить команду;
* action – идентификатор действия, которое описывает данную команду;
* additional\_information – дополнительная информация, которая различается в зависимости от того какая команда исполняется в данный момент;

Идентификаторы целевых блоков описаны в виде набора констант в модуле target\_block\_types:

SETTINGS\_BLOCK = 0x001

EVENT\_LOG\_BLOCK = 0x002

RUSES\_MANAGEMENT\_BLOCK = 0x003

MONITOR\_INTERACTION\_BLOCK = 0x004

Идентификаторы действий также описаны в виде набора констант модуля action\_types:

CREATE = 0x001

READ = 0x002

UPDATE = 0x003

DELETE = 0x004

EXECUTE = 0x005

RESTORE = 0x006

EXPORT = 0x007

IMPORT = 0x008

START = 0x009

STOP = 0x00A

Интерфейс CommandsDescriptionInterface реализован классом CommandsDescriptionModule в модуле commands\_description. Данный класс содержит в себе таблицу отношений между командой и её обработчиком, которая фактически описывает API приложения. Ключом данной таблицы является пара идентификаторов: идентификатор целевого блока и идентификатор действия. В обработчике разбираются аргументы команды, и она делегируется модулю, который должен её выполнить. Список обработчиков команд, которые могут быть выполнены приложением выглядит следующим образом:

* \_create\_setting\_handler – обработчик команды задания новой настройки приложения. В качестве входного параметра принимает объект настройки (пара ключ\значение);
* \_read\_settings\_handler – обработчик команды чтения настройки приложения. Возвращаемое значение зависит от входного параметра: если входной параметр не определён – возвращается список всех настроек, если входной параметр является списком ключей – возвращается список настроек по списку, если входной параметр является одиночным ключом – возвращается значение по этому ключу;
* \_update\_setting\_handler – обработчик команды обновления ранее созданной настройки. В качестве входного параметра принимает новый объект настройки;
* \_delete\_setting\_handler – обработчик команды удаления существующей настройки. В качестве входного параметра принимает ключ настройки;
* \_import\_settings\_handler – обработчик команды импорта настроек. В качестве входного параметра принимает путь к внешнему файлу с настройками;
* \_export\_settings\_handler – обработчик команды экспорта настроек. В качестве входного параметра принимает путь к целевому файлу, в который приложение запишет настройки.
* \_read\_event\_log\_handler – обработчик команды чтения журнала событий файловой системы. В качестве входного параметра передаётся промежуток дат. Возвращает события, произошедшие в системе за этот промежуток;
* \_restore\_state\_by\_event\_log\_handler – обработчик команды восстановления файловой системы по журналу событий. В качестве входного параметра принимает либо идентификатор события, либо промежуток идентификаторов. Изменения, вызванные событием, либо набором событий по возможности откатываются приложением в исходное состояние;
* \_create\_rule\_handler – обработчик команды создания правила. В качестве входного параметра принимает объект правила;
* \_read\_rules\_handler – обработчик команды чтения всех правил, заданных в приложении. Возвращает коллекцию правил;
* \_update\_rule\_handler – обработчик команды обновления правила. В качестве входного параметра принимает объект правила;
* \_delete\_rule\_handler – обработчик команды удаления правила. В качестве входного параметра принимает идентификатор правила;
* \_import\_rules\_handler – обработчик команды импорта правил. В качестве входного параметра принимает путь к внешнему файлу с правилами;
* \_export\_rules\_handler – обработчик команды экспорта правил. В качестве входного параметра принимает путь к целевому файлу;
* \_execute\_action\_handler – обработчик команды немедленного выполнения действия над указанным файлом. В качестве входного параметра принимает путь к файлу и объект действия, который будет рассмотрен позднее;
* \_start\_monitor\_handler – обработчик команды запуска монитора;
* \_stop\_monitor\_handler – обработчик команды остановки монитора;

Если обработчик команды не найден по ключу – генерируется исключение NotImplementedError, информация об исключении записывается в журнал и приложение продолжает работать в обычном режиме.

## **3.3** Модуль журналирования

Модуль журналирования предназначен для записи отчётов о всех действиях приложения в файловой системе пользователя в базу данных, а также для записи в файл журнала всех отладочных сообщений и сообщений об ошибках и предупреждениях. Модуль журналирования представлен в приложении классом LoggerModule:

class LoggerModule(metaclass=Singleton):

def \_\_init\_\_(self, filename, database):

pass

def write\_event\_info(self, log\_record):

pass

def read\_event\_info(self, log\_record\_id):

pass

def read\_events\_info\_range(self, start\_record\_id, end\_record\_id):

pass

def read\_events\_info\_daterange(self, start\_date, end\_date):

pass

@property

def logger(self):

pass

Методы write\_event\_info и read\_event\_info предназначены для записи и чтения события в файловой системе соответственно. Методы read\_events\_info\_range и read\_events\_info\_daterange позволяют читать данные по промежуткам идентификаторов и дат создания записей.

Свойство logger предоставляет доступ к объекту класса Logger встроенного в язык Python модуля logging, который позволяет настроить журналирование строк информации по типам в файл или консоль, настроить формат записей, задать обработчики для разных типов записей и т.д.

Формат записи в файле журнала будет выглядеть следующим образом:

'%(levelname)s [%(asctime)s]: %(message)s'

Так же стоит отметить что модуль журналирования может быть создан только в одном экземпляре, что описывается метаклассом (классом который управляет созданием объектов другого класса) Singleton:

class Singleton(type):

""" Use to create a singleton object.

"""

def \_\_init\_\_(cls, name, bases, dict):

super().\_\_init\_\_(name, bases, dict)

cls.\_instance = None

def \_\_call\_\_(cls, \*args, \*\*kwargs):

if cls.\_instance is None:

cls.\_instance = super(Singleton,

cls).\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instance

Логика метода \_\_call\_\_ запускается в момент создания объекта класса, к которому добавлен вышеописанный метакласс и не даёт создать объект cls.\_instance если он уже был ранее создан.

## **3.4** Модуль настройки приложения

Модуль настройки приложения предоставляет доступ к файлу настроек как к словарю значений по ключам. Так же он предоставляет возможность импортировать и экспортировать файл настроек приложения во время его работы. Модуль настройки представлен классом SettingsModule, экземпляр которого, также, как и экземпляр LoggerModule, можно создать лишь один на приложение:

class SettingsModule(metaclass=Singleton):

def \_\_init\_\_(self):

pass

def \_\_getitem\_\_(self, key):

pass

def \_\_setitem\_\_(self, key, value):

pass

def \_\_delitem\_\_(self, key):

pass

def get\_all\_settings(self):

pass

def get\_settings\_by\_keys(self, keys\_list):

pass

def export\_settings(self, destination\_path):

pass

def import\_settings(self, source\_path):

pass

Методы \_\_getitem\_\_, \_\_setitem\_\_, \_\_delitem\_\_ добавляют объекту класса функциональность индексатора, то есть дают возможность обращаться к настройкам, инкапсулированным классом, следующим образом:

* value = obj[key] – чтение настройки по ключу;
* obj[key] = value – запись настройки по ключу;
* del obj[key] – удаление настройки по ключу;

Метод get\_all\_settings позволяет получить весь список настроек, что полезно для обеспечения доступа к ним через интерфейс пользователя. Метод get\_settings\_by\_keys принимает на вход список ключей и возвращает настройки по этим ключам, что позволяет на стороне интерфейса пользователя разделить настройки на секции.

Метод export\_settings и import\_settings принимают в качестве входных параметров путь к файлу для экспорта и путь к файлу для импорта соответственно. Экспорт настроек побочных эффектов не имеет, но после импорта нужно перезапустить приложение для того что бы все настройки применились.

## **3.5** Модуль хранения данных

Модуль хранения данных предоставляет остальному приложению простые и надёжные интерфейсы по работе с базой данных MongoDB. База MongoDB была выбрана исходя из того, что она обладает высокой скоростью записи и нефиксированной структурой документов. Это позволяет хранить в коллекции документы (BSON объекты) с различающимся набором полей, что очень важно при хранении правил и записей журнала, где данные могут различаться в зависимости от сложности правила или типа произошедшего события.

Интерфейс базы данных предоставляет стандартные CRUD-операции, и объявлен в виде класса EntityStorageInterface:

class EntityStorageInterface(metaclass=ABCMeta):

@abstractmethod

def create(entity):

pass

@abstractmethod

def read(entity\_id):

pass

@abstractmethod

def update(entity):

pass

@abstractmethod

def delete(entity\_id):

pass

@abstractmethod

def read\_all():

pass

Этот интерфейс имеет 2 реализации: EntityStorage и CachedEntityStorage. Отличаются они тем, что в реализации CachedEntityStorage результат вызова методов read и read\_all кэшируются. Методы create, update, delete работают так же, как и аналогичные методы EntityStorage с той лишь разницей, что перед тем как создать изменить или удалить сущность в базе данных они должны сделать изменения в массиве закэшированных объектов. Реализация интерфейса хранения с кэшем нужна для модуля управления правилами, потому что при большом потоке событий от файловой системы неприемлемо каждый раз читать правила из базы данных.

## **3.6** Модуль анализа файлов

Модуль анализа файлов предназначен для сбора всей статистики по файлу, для которого произошло событие в файловой системе. Модуль представлен классом FileAnalyzer, имеющим следующий интерфейс:

class FileAnalyzer:

def \_\_init\_\_():

pass

@staticmethod

def analyse\_file(file\_path):

pass

Интерфейс класса содержит единственный метод, который принимает путь к файлу и возвращает объект с информацией о нём. Объект этот представлен в приложении классом AnalysisResult:

class AnalysisResult:

def \_\_init\_\_(self):

self.\_is\_directory = False

self.\_name = None

self.\_extension = None

self.\_content\_type = None

self.\_size = None

self.\_content\_specific\_info = None

@property

def is\_directory(self):

return self.\_is\_directory

@is\_directory.setter

def is\_directory(self, value):

self.\_is\_directory = value

@property

def name(self):

return self.\_name

@name.setter

def name(self, value):

self.\_name = value

@property

def extension(self):

return self.\_extension

@extension.setter

def extension(self, value):

self.\_extension = value

@property

def content\_type(self):

return self.\_content\_type

@content\_type.setter

def content\_type(self, value):

self.\_content\_type = value

@property

def size(self):

return self.\_size

@size.setter

def size(self, value):

self.\_size = value

@property

def content\_specific\_info(self):

return self.\_content\_specific\_info

@content\_specific\_info.setter

def content\_specific\_info(self, value):

self.\_content\_specific\_info = value

Данный класс содержит следующие свойства:

* is\_directory – определяет директория это или обычный файл;
* name – имя файла либо директории;
* extension – расширение файла, имеет значение None в случае директории;
* content\_type – идентификатор типа содержимого файла;
* size – размер файла
* content\_specific\_info – информация, специфичная для определённого типа содержимого;

Стоит отдельно обратить внимания на два свойства: content\_type на и content\_specific\_info. Наличие этих двух свойств в результате анализа файла позволят пользователю задавать правила, касающиеся не только файлов с определённым расширением или именем, но файлов с определённым с определённым типом контента. Типы контента доступные для задания в правилах и обнаруживаемые модулем анализа файлов описаны в виде набора констант в модуле content\_types:

ARCHIVE = 0x001

NOTE = 0x002

DOCUMENT = 0x003

BOOK = 0x004

AUDIO = 0x005

VIDEO = 0x006

BINARY = 0x007

SYSTEM = 0x008

IMAGE = 0x009

Информация, предоставляемая свойством content\_specific\_info зависит от типа контента и описывается следующей иерархией классов:

* ContentSpecificInfo – класс, являющийся общим предком всех классов для специфичной по типу контента информации. Содержит идентификатор типа контента, описанный в модуле content\_types.
* MusicSpecificInfo – наследник ContentSpecificInfo, содержит в себе свойства, позволяющие получить информацию о музыкальных файлах: название группы, жанр музыки и названии альбома;
* TextSpecificInfo – наследник ContentSpecificInfo, содержит в себе свойства, предоставляющие информацию о теме и языке текстового документа;
* DocumentSpecificInfo – наследник TextSpecificInfo, не содержит дополнительной информации относительно базового класса;
* BookSpecificInfo – наследник TextSpecificInfo, содержит свойства, специфичные для книг и позволяющие кроме темы и языка получить информацию о авторе и жанре книги;

Более подробное определение этих классов смотрите в приложении Ж.

## **3.7** Модуль пересылки сообщений

Приложение для данного дипломного проекта состоит из трёх отдельных программ: программа мониторинга файловой системы, программа анализа и распределения файлов, программа-клиент. Модуль пересылки сообщений отвечает за обмен данными между этими тремя программами.

Данные передаются в виде текстовых строк, в формате, который заранее описан протоколом между отправителем и получателем. В качестве инструмента для передачи сообщений используется библиотека ZeroMQ, предоставляющая надёжный интерфейс для обмена сообщениями через сокеты и позволяющая быстро построить различные шаблоны взаимодействия, такие как точка-точка, издатель-подписчик и т.д.

Модуль пересылки сообщений имеет следующий интерфейс:

class MessagingInterface(metaclass=ABCMeta):

@abstractmethod

def send\_message(self, message):

pass

@abstractmethod

def receive\_message(self):

pass

Реализует интерфейс обмена сообщениями класс MessagingManager, который описан следующим образом:

class MessagingManager(MessagingInterface):

def \_\_init\_\_(self, messaging\_manager\_type,\

socket\_binding\_address):

self.\_send\_queue = Queue()

self.\_receive\_queue = Queue()

self.\_background\_worker\_stop\_event = Event()

self.\_background\_worker = None

def \_\_del\_\_(self):

pass

def send\_message(self, message):

pass

def is\_received\_messages\_exists(self):

pass

def receive\_message(self):

pass

def get\_all\_received\_messages(self):

pass

При инициализации объекта класса в метод конструктора \_\_init\_\_ передаётся тип менеджера пересылки сообщений – клиент или сервер, и адрес, к которому будет привязан сокет для обмена сообщениями. Поэтому для того что бы создать соединение, каждая из программ, участвующих в обмене, должна создать объект своего типа (зависит от назначения программы) и передать в качестве параметра конструктора один и тот же адрес. Создание пары таких объектов выглядит следующим образом:

server = MessagingManager(MessagingManagerType.SERVER,

"tcp://127.0.0.1:5555")

client = MessagingManager(MessagingManagerType.CLIENT,

"tcp://127.0.0.1:5555")

Также при инициализации создаются две очереди класса Queue для принятых и переданных сообщений, объект события \_background\_worker\_stop\_event класса Event, а также объект класса MessagingManagerBackgroudReciever, который представляет собой поток Thread основной задачей которого является приём приходящих и отправка исходящих сообщений в бесконечном цикле. Объект события нужен для того, чтобы при удалении сборщиком мусора объекта менеджера пересылки сообщений (и соответственно вызова деструктора \_\_del\_\_), можно было остановить поток приёма-передачи сообщений.

Передача данных между двумя потоками идёт через потокобезопасные очереди \_send\_queue и \_receive\_queue. При вызове метода send\_message, в который в качестве параметра передаётся строка сообщения, объект класса MessagingManager не отправляет сообщение не передаёт сообщение непосредственно, а кладёт его в очередь \_send\_queue. Методы receive\_message и get\_all\_received\_messages возвращают сообщения, читая их из очереди \_receive\_queue. Таким образом, использующая эту функциональность программа получает асинхронную реализацию приёма\передачи сообщений.

Метод is\_received\_messages\_exists предназначен для проверки очереди на наличие принятых сообщений. Он используется в тех случаях, когда код, принимающий сообщения в бесконечном цикле, ждёт их поступления.

Класс MessagingManagerBackgroudReciever описан следующим образом:

class MessagingManagerBackgroudReciever(Thread):

def \_\_init\_\_(self, messaging\_manager\_type, \

socket\_binding\_address, queues, events):

super().\_\_init\_\_()

send\_queue, recieve\_queue = None

stop\_event = None

self.\_send\_queue = None

self.\_recieve\_queue = None

self.\_stop\_event = None

self.\_context = None

self.\_socket = None

def \_bind\_socket\_to\_address(self,\

messaging\_manager\_type, binding\_address):

pass

def run(self):

pass

При создании объекта этого класса создаётся контекст ZeroMQ, с помощью этого контекста создаётся сокет, предназначенный для пересылки сообщения. Код создания сокета следующий:

self.\_socket = self.\_context.socket(zmq.PAIR)

Константа zmq.PAIR определяет для сокета шаблон взаимодействия точка-точка - это значит, что участников передачи может быть максимум двое и они равноправны. Но несмотря на то что участники передачи равноправны, интерфейс сокетов ZeroMQ спроектирован так что один из сокетов должен быть привязан к адресу с помощью метода bind и являться условным сервером, а второй с помощью метода connect и являться условным клиентом. Разделяются они с помощью параметра конструктора messaging\_manager\_type который определяется как перечисление:

class MessagingManagerType(Enum):

CLIENT = 1

SERVER = 2

За привязку сокета к адресу отвечает метод \_bind\_socket\_to\_address, который вызывает методы сокета bind или connect в зависимости от типа менеджера пересылки сообщений.

Метод run запускается при старте потока и обслуживает очереди входящих и исходящих сообщений в цикле. Цикл завершается, когда объект события \_stop\_event оказывается в активном состоянии. После того как цикл завершился объект сокета закрывается, а объект контекста уничтожается.

## **3.8** Модуль управления правилами.