

### 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 модулей:

- 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 исполняемом модуле `terminal_client`. Функции, отвечающие за его реализацию:

- 1) `_parse_args` – функция, которая не принимает ничего и возвращающая словарь с полученными от пользователя аргументами и

флагами. Аргументы командной строки, введенные пользователем, достаются стандартным модулем `argparse` из коллекции `sys.argv`, после чего преобразуются в вид словаря по заранее заданным правилам.

- 2) `_get_command_json_string` – функция, которая принимает на вход словарь с аргументами и возвращающая команду, сериализованную в строку JSON.
- 3) `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 исполняемом 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:

- 1) `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.
- 2) `_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 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:

- 1) `_create_setting_handler` – обработчик команды задания новой настройки приложения. В качестве входного параметра принимает объект настройки (пара ключ\значение);
- 2) `_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;
- 3) `_update_setting_handler` – обработчик команды обновления ранее созданной настройки. В качестве входного параметра принимает новый объект настройки;
- 4) `_delete_setting_handler` – обработчик команды удаления существующей настройки. В качестве входного параметра принимает ключ настройки;
- 5) `_import_settings_handler` – обработчик команды импорта настроек. В качестве входного параметра принимает путь к внешнему файлу с настройками;
- 6) `_export_settings_handler` – обработчик команды экспорта настроек. В качестве входного параметра принимает путь к целевому файлу, в который приложение запишет настройки.
- 7) `_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;
- 8) `_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;

- 9) `_create_rule_handler` – обработчик команды создания правила. В качестве входного параметра принимает объект правила;
- 10) `_read_rules_handler` – обработчик команды чтения всех правил, заданных в приложении. Возвращает коллекцию правил;
- 11) `_update_rule_handler` – обработчик команды обновления правила. В качестве входного параметра принимает объект правила;
- 12) `_delete_rule_handler` – обработчик команды удаления правила. В качестве входного параметра принимает идентификатор правила;
- 13) `_import_rules_handler` – обработчик команды импорта правил. В качестве входного параметра принимает путь к внешнему файлу с правилами;
- 14) `_export_rules_handler` – обработчик команды экспорта правил. В качестве входного параметра принимает путь к целевому файлу;
- 15) `_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;
- 16) `_start_monitor_handler` – обработчик команды запуска монитора;
- 17) `_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):
```



```
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.

Поскольку база данных не реляционная, и таблицы со связями в ней отсутствуют, привести схему данных не представляется возможным. Но поскольку в базе данных, используемой описываемым приложением только две коллекции с данными: коллекция информации о действиях, произведенных приложением над файлами и коллекция правил, можно привести пример типовых документов, которые будут в этих коллекциях храниться. Пример документа, хранящего в себе информацию о удалении приложением файла:

```
{
  "_id": ObjectId("12edf42342fgg"),
  "target_file": "/home/username/temp/file.txt",
  "action": {
    "action_type": 1
    "is_permanent_deleting": FALSE
  }
}
```

Как можно увидеть далее, документ, который храниться в базе, полностью совпадает по набору свойств с объектом языка Python, что позволяет заметно



упростить разработку и убрать расход времени на сборку объекта по частям, как это было бы в случае работы с реляционными базами данных. Документ считывается из базы данных и напрямую отображается в объект Python.

Пример документа из коллекции правил:

```
{
  "_id": ObjectId("12edf42342fgg"),
  "target_directory": "/home/username/temp/",
  "file_constraints": {
    "is_directory": FALSE,
    "target_event_types": [1, 2],
    "target_content_types": [1, 2],
    "target_name_template": Null,
    "target_extention_template": Null,
    "target_file_max_size": Null
    "target_file_min_size": Null
  },
  "action": {
    "action_type": 1,
    "is_permanent_deleting": FALSE
  }
}
```

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:

- 1) `ContentSpecificInfo` – класс, являющийся общим предком всех классов для специфичной по типу контента информации. Содержит идентификатор типа контента, описанный в модуле `content_types`.
- 2) `MusicSpecificInfo` – наследник `ContentSpecificInfo`, содержит в себе свойства, позволяющие получить информацию о музыкальных файлах: название группы, жанр музыки и названии альбома.
- 3) `TextSpecificInfo` – наследник `ContentSpecificInfo`, содержит в себе свойства, предоставляющие информацию о теме и языке текстового документа.
- 4) `DocumentSpecificInfo` – наследник `TextSpecificInfo`, не содержит дополнительной информации относительно базового класса.
- 5) `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.

Модуль управления правилами является одним из ключевых модулей приложения и отвечает за всю логику работы с правилами распределения файлов. Правило распределения файлов, в свою очередь является ключевой сущностью и представлено классом `Rule`:

```
class Rule:
    def __init__(self, target_directory, file_constraints, \
                  action):
        self._target_directory = target_directory
        self._file_constraints = file_constraints
        self._action = action

    @property
    def target_directory(self):
        return self._target_directory

    @property
    def file_constraints(self):
        return self._file_constraints

    @property
    def action(self):
        return self._action
```

Класс правила представлен всего тремя свойствами:

- 1) Свойство `target_directory` представляет собой путь к директории и позволяет определить, к файлам какой отслеживаемой директории это правило относится.
- 2) Свойство `file_constraints` задаёт список ограничений, которым должен соответствовать файл что бы к нему было применено это правило. Значение свойства представлено классом `RuleFileConstraint`.
- 3) Свойство `action` определяет действие, которое применяется к файлу, удовлетворяющему условию этого правила. Значение свойства представлено классом-наследником класса `RuleAction`.

Определение класса, представляющего информацию о ограничениях



целевого файла - RuleFileConstraint:

```
class RuleFileConstraint:
    def __init__(self):
        self._is_directory = False
        self._target_event_types = []
        self._target_content_types = []
        self._target_name_template = None
        self._target_extension_template = None
        self._target_file_max_size = None
        self._target_file_min_size = None
```

Список ограничений, которые могут быть определены пользователем и применены к файлу:

- 1) `is_directory` – если значение свойства истинно – правило применяется только к директориям. Если ложно – и к директориям, и к файлам.
- 2) `target_event_types` – свойство представляющее собой массив событий. Если событие, произошедшее с файлом, не входит в этот массив – правило к нему не применяется.
- 3) `target_content_types` – свойство представляющее собой массив типов контента. Если тип контента файла не подходит – правило не применяется.
- 4) `target_name_template` – представляет собой шаблон регулярного выражения. Имя файла должно совпадать с этим шаблоном.
- 5) `target_extension_template` – также шаблон регулярного выражения. Расширение файла должно подпадать под этот шаблон.
- 6) `target_file_max_size` – ограничение по максимальному размеру файла.
- 7) `target_file_min_size` – ограничение по минимальному размеру файла.

Не обязательно определять все из вышеперечисленных ограничений при создании правила, но всем, что были определены, информация о файле и информация о событии, произошедшем с файлом, должны соответствовать.

Информация о действии, представленная свойством `action` представлена иерархией классов, каждый класс в которой описывает конкретное действие и свойства специфичные для выполнения этого из действия. Иерархия классов выглядит следующим образом:

- 1) `RuleAction` – базовый класс для всех действий в правиле. Содержит идентификатор действия.
- 2) `DeleteFileRuleAction` – класс, представляющий удаление файла.

Содержит свойство, предоставляющее информацию о том, нужно ли удалять файл навсегда или его стоит переместить в корзину.

- 3) `ReplaceFileRuleAction` – класс, представляющий перемещение файла. Поскольку файл может быть перемещён как в уже существующую директорию, так и в директорию, которая может быть создана на основе его имени – класс содержит свойства, задающие путь к целевой директории либо шаблон имени целевой директории.
- 4) `RenameFileRuleAction` – класс, представляющий переименование файла. Содержит свойство, задающее шаблон нового имени файла.
- 5) `GroupByAttributeFileRuleAction` – класс, описывающий действие по группировке файлов по определённому признаку. Признак задаётся типом действия. Содержит в себе свойство, задающее шаблон целевой директории.

Идентификатор действия определён в виде набора констант в модуле `rule_action_types`:

```
# common actions
DELETE_FILE           = 0x001
REPLACE_FILE          = 0x002
RENAME_FILE           = 0x003
IGNORE                = 0x00B
# music actions
GROUP_BY_MUSIC_BAND   = 0x004
GROUP_BY_MUSIC_GENRE   = 0x005
GROUP_BY_MUSIC_ALBUM   = 0x006
# books and docs actions
GROUP_BY_SUBJECT       = 0x007
GROUP_BY_LANGUAGE      = 0x008
GROUP_BY_GENRE          = 0x009
GROUP_BY_AUTHOR        = 0x00A
```

Модуль, позволяющий работать с вышеописанными правилами, реализован в виде класса `RulesManager`:

```
class RulesManager(metaclass=Singleton):
    def __init__(self):
        pass

    def create_rule(self, rule):
        pass

    def update_rule(self, rule):
        pass
```

```

def delete_rule(self, rule_id):
    pass

def read_all_rules(self):
    pass

def get_rules(self, event_info, analyze_results=None):
    pass

def export_rules(self, target_file):
    pass

def import_rules(self, source_file):
    pass

def add_directory_list_change_handlers(self,
    add_callback, remove_callback):
    pass

```

Методы `create_rule`, `update_rule`, `delete_rule` и `read_all_rules` делегируют вызовы объекту класса `CachedEntityStorage`, напрямую передавая им свои входные параметры. Кроме того, в методах `create/update/delete_rule` отслеживается появление новых или полное удаление старых отслеживаемых директорий.

Метод `get_rules` принимает информацию о событии в файловой системе и опциональный результат анализа файла и возвращает все правила, соответствующие этому событию.

Методы `export_rules` и `import_rules` предназначены для экспорта и импорта правил в систему соответственно, пути к необходимым для этого файлам передаются в качестве параметра метода. Данные импортируются и экспортируются в формате JSON.

Метод `add_directory_list_change_handlers` позволяет добавить два обработчика, один из которых сработает если список директорий, которые отслеживаются приложением – увеличился, другой – если он уменьшился. К примеру: пользователь добавил правило для новой директории или удалил правило, которое является последним для определённой директории. И в том и в другом случае программа-монитор должна быть оповещена, чтобы обновить список отслеживаемых директорий и не генерировать бесполезные события.

### 3.9 Модуль взаимодействия с монитором

Модуль взаимодействия с монитором является центральным модулем приложения. Его главной задачей является обработка сообщений, приходящих от

монитора файловой системы и передача сообщений от других модулей, в частности от модуля управления правилами и от модуля описания команд, монитору. Реализация модуля представлена двумя классами: MonitorMessagesProcessingLoop и MonitorInteractionManager:

```
class MonitorMessagesProcessingLoop(Thread):
    def __init__(self, messaging_manager, stop_event):
        pass

    def _process_messages(self, messages):
        pass

    def run(self):
        pass

class MonitorInteractionManager(metaclass=Singleton):
    def __init__(self):
        self._send_queue = None
        self._receive_queue = None
        self._logger_module = None
        self._settings_module = None
        self._messaging_manager = None
        self._rules_manager = None
        self._stop_event = None
        self._processing_loop = None

    def __del__():
        pass

    def _add_watching_directory_handler(self,
        directory_path):
        pass

    def _remove_watching_direcotory_handler(self,
        directory_path):
        pass

    def rollback_actions(self, event_log_records):
        pass

    def execute_action_immediately(self, file_path, action):
        pass

    def start_monitor(self):
        pass
```

```
def stop_monitor(self):  
    pass
```

Класс `MonitorMessagesProcessingLoop` представляет собой объект потока, который в бесконечном цикле в методе `run` обменивается сообщениями с программой монитором. Метод `_process_messages` вызывается из метода `run` и создаёт новый поток обработки для каждого из пришедших от монитора сообщений.

Класс `MonitorInteractionManager` агрегирует в себе реализации большинства ранее описанных модулей, а также содержит две потокобезопасные очереди для обмена сообщениями и объект потока обработки `_processing_loop` и событие для остановки этого потока в случае удаления менеджера - `_stop_event`, которое устанавливается в деструкторе `__del__`.

Метод `_add_watching_directory_handler` служит как функция-обработчик, которая устанавливается через модуль управления правилами и срабатывает при добавлении новой отслеживаемой директории. То же самое, но для удаления директории – метод `_remove_watching_directory_handler`.

Метод `execute_action_immediately` позволяет выполнить заданное пользователем действия, не смотря ни на какие правила немедленно. Этот метод полезен в случае если пользователю придётся вручную откатывать последствия некоторых операций. Что бы какое-то из заданных пользователем правил, действие которого было для него неприемлемым не применилось снова, перед тем как исполнить над файлом заданное пользователем действие для него добавляется правило с действием `IGNORE`, которое отменяет все остальные.

Методы `start_monitor` и `stop_monitor` предназначены для передачи команды монитору о старте и остановке соответственно. Это может быть полезно в ситуациях, когда пользователя устраивает работа приложения, но ему нужно скопировать большое число файлов в отслеживаемую директорию, и он знает, что работа монитора скажется на производительность персонального компьютера. В таком случае монитор останавливается пользователем в целях оптимизации и запускается после завершения действия.

### 3.10 Модуль мониторинга файловой системы

Данный модуль предназначен для отслеживания и передачи сообщений о событиях, происходящих в отслеживаемых директориях. Функциональность монитора реализована в виде исполняемого модуля `monitor`. Монитор имеет следующий набор функций:

- 1) `main` – главная функция модуля, в которой создаётся контекст `Inotify`

(библиотеки, отвечающей за отслеживание событий), и запускается цикл, в котором по порядку обслуживаются сначала приходящие сообщения, затем события от файловой системы.

- 2) `process_recieved_messages` – функция, предназначенная для приёма и обработки принятых сообщений. Принимает в качестве параметров контекст `Inotify`, и список сообщений. С помощью контекста удаляются и добавляются отслеживаемые директории.
- 3) `process_recieved_events` – функция, принимающая в качестве параметров объект контекста и менеджер пересылки сообщений. Считывает несколько сообщений из буфера контекста, конвертирует их в сообщения и отправляет модулю взаимодействия с монитором.
- 4) `process_possible_pair_event` – функция, которая обрабатывает ситуацию, когда событие может быть парным. Например, переименование файла это два события, потому что это фактически перемещение файла в файл с другим именем. Функция принимает на вход текущее событие и список с возможно-парными событиями.
- 5) `get_message_by_event_pair` – функция, которая принимает парное событие и возвращает объект сообщения для этого события.
- 6) `get_message_by_event` – функция, которая принимает событие и возвращает объект сообщения для этого события.
- 7) `get_event_type_by_mask` – функция, позволяющая перевести флаг события `Inotify` во внутренний для приложения тип события.

Тип события от монитора в приложении представлен набором констант в модуле `monitor_event_types`:

<code>FILE_CREATED</code>	<code>= 0x001</code>
<code>FILE_NAME_CHANGED</code>	<code>= 0x002</code>
<code>FILE_CONTENT_CHANGED</code>	<code>= 0x004</code>
<code>FILE_INCLUDED</code>	<code>= 0x008</code>
<code>FILE_EXCLUDED</code>	<code>= 0x010</code>
<code>FILE_DELETED</code>	<code>= 0x020</code>
<code>DIRECTORY_REPLACED</code>	<code>= 0x040</code>
<code>DIRECTORY_DELETED</code>	<code>= 0x080</code>
<code>FILE_METADATA_CHANGED</code>	<code>= 0x100</code>

Объект сообщения, который передаётся от монитора и к монитору описан классом `MonitorMessage`:

```
class MonitorMessage:
    def __init__(self):
        self._target_directory = ""
        self._target_file = ""
```

```

        self._additional_information = ""
        self._event_type = 0

    def __repr__(self):
        return self.__combine_message()

    def __str__(self):
        return self.__combine_message()

    def __combine_message(self):
        pass

    @staticmethod
    def restore_from_string(message):
        pass

```

Сообщение состоит из следующих частей:

- `target_directory` – имя отслеживаемой директории;
- `target_file` – имя файла, с которым случилось событие;
- `additional_information` – дополнительная информация, например, новое имя файла при переименовании;
- `event_type` – тип события от монитора, представляет собой одну из констант, описанных выше;

Метод `__combine_message` позволяет преобразовать объект в строку, где значения свойств разделяются двоеточиями. В этом формате сообщение пересылается модулю взаимодействия с монитором. Методы `__repr__` и `__str__` позволяют привести объект к строковому формату, что позволяет распечатывать его функцией `print` и переводить в строку функцией `str`. Метод `restore_from_string` используется на принимающей стороне для восстановления сообщения в объектный вид.

В данном разделе были проанализированы все модули из которых состоит приложение данного дипломного проекта, описаны все функции, классы и списки констант, детально описано взаимодействие между модулями и форматы передачи данных. Детальную диаграмму классов приложения можно увидеть на (ссылка на и диаграмму классов).