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 следующими классами:

- `MusicSpecificInfo` – содержит в себе свойства, позволяющие получить информацию о музыкальных файлах: название группы, жанр музыки и названии альбома;
- `DocumentSpecificInfo` – содержит свойства, позволяющие получить информацию о документах: тема документа и язык, на котором он написан;
- `BookSpecificInfo` – 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.