# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignorin# [file name]: session\_manager.py (УПРОЩЕННАЯ ВЕРСИЯ)

import os

import uuid

from datetime import datetime, timedelta nimport os import fnmatch import re from typing import List from shared.logger import setup\_logger class FileValidator: def \_\_init\_\_(self, config: dict): self.config = config self.logger = setup\_logger(\_\_name\_\_) # УНИВЕРСАЛЬНАЯ КЛАССИФИКАЦИЯ ФАЙЛОВ - РАСШИРЕННАЯ ВЕРСИЯ self.FILE\_CATEGORIES = { # ОСНОВНЫЕ ФАЙЛЫ - отслеживаем сессии 'MAIN': [ # Office '.doc', '.docx', '.xls', '.xlsx', '.ppt', '.pptx', '.rtf', # PDF и текстовые '.pdf', '.txt', '.md', # OpenDocument '.odt', '.ods', '.odp', # CAD системы '.dwg', '.dxf', '.dgn', '.rvt', '.rfa', '.rte', '.sat', '.ipt', '.iam', '.prt', '.asm', '.sldprt', '.sldasm', '.3dm', '.skp', '.max', '.blend', '.mb', '.ma', # Credo и геодезические системы '.crproj', '.credoproj', '.gpx', '.kml', '.kmz', # Архивы и образы '.zip', '.rar', '.7z', '.iso', # Медиа (для полноты) '.jpg', '.jpeg', '.png', '.bmp', '.tiff', '.tif' ], # ВРЕМЕННЫЕ ФАЙЛЫ - игнорируем для сессий, но отслеживаем для контекста 'TEMPORARY': [ # Общие временные '.tmp', '.temp', '.crdownload', '.part', # Office временные '~$', '~wr', '~wrd', '~wrl', '~rf', # CAD временные и backup '.bak', '.dwl', '.dwl2', '.sv$', '.autosave', # Системные временные '.lock', '.lck' ], # ПОЛНОСТЬЮ ИГНОРИРУЕМЫЕ ФАЙЛЫ - даже не обрабатываем события 'IGNORE': [ '.log', '.cache', '.DS\_Store', '.thumb', '.thumbs', 'desktop.ini', '.tmp.metadata' ] } # ДОБАВЛЕНО: Стандартные имена Office файлов для специальной обработки self.OFFICE\_DEFAULT\_NAMES = [ 'новый документ microsoft word.docx', 'новый документ microsoft word.doc', 'новая книга microsoft excel.xlsx', 'новая книга microsoft excel.xls', 'новая презентация microsoft powerpoint.pptx', 'новая презентация microsoft powerpoint.ppt', 'document.docx', 'document.doc', 'workbook.xlsx', 'workbook.xls', 'presentation.pptx', 'presentation.ppt', 'лист microsoft excel.xlsx', 'лист microsoft excel.xls', 'документ microsoft word.docx', 'документ microsoft word.doc' ] # Компилируем паттерны для лучшей производительности self.ignore\_patterns = self.config.get('ignore\_patterns', []) self.ignore\_extensions = self.config.get('ignore\_extensions', []) self.ignore\_dirs = self.config.get('ignore\_dirs', []) # ДОБАВЛЕНО: Кэш для быстрого определения категорий self.\_category\_cache = {} # ДОБАВЛЕНО: Расширенные паттерны для временных файлов self.\_extended\_temp\_patterns = [ # Файлы без расширения с HEX-именами (типичные временные файлы) r'^[0-9A-F]{4,16}$', r'^[0-9A-F]{4,16}\.tmp$', r'^[0-9A-F]{4,16}\.temp$', # Файлы с короткими именами (часто временные) r'^[A-Z0-9]{4,8}$', r'^[A-Z0-9]{4,8}\.tmp$', ] self.logger.info(f"✅ FileValidator initialized with {len(self.FILE\_CATEGORIES['MAIN'])} main formats, " f"{len(self.FILE\_CATEGORIES['TEMPORARY'])} temporary patterns, " f"{len(self.FILE\_CATEGORIES['IGNORE'])} ignored patterns") def get\_file\_category(self, file\_path: str) -> str: """Определяет категорию файла""" filename = os.path.basename(file\_path) # Проверяем кэш if file\_path in self.\_category\_cache: return self.\_category\_cache[file\_path] # 1. Проверяем полностью игнорируемые файлы if self.\_is\_ignored\_file(filename): self.\_category\_cache[file\_path] = 'IGNORE' return 'IGNORE' # 2. Проверяем временные файлы (включая расширенные паттерны) if self.\_is\_temporary\_file(filename): self.\_category\_cache[file\_path] = 'TEMPORARY' return 'TEMPORARY' # 3. Проверяем основные файлы file\_ext = os.path.splitext(filename)[1].lower() if file\_ext in self.FILE\_CATEGORIES['MAIN']: self.\_category\_cache[file\_path] = 'MAIN' return 'MAIN' # 4. Файл не подходит ни под одну категорию - считаем IGNORE self.\_category\_cache[file\_path] = 'IGNORE' return 'IGNORE' def is\_office\_default\_name(self, file\_path: str) -> bool: """Проверяет является ли файл стандартным именем Office""" filename = os.path.basename(file\_path).lower() return filename in self.OFFICE\_DEFAULT\_NAMES def should\_monitor\_file(self, file\_path: str) -> bool: """Определяет нужно ли отслеживать файл""" if not os.path.isfile(file\_path): return False # Используем универсальную классификацию file\_category = self.get\_file\_category(file\_path) if file\_category == 'IGNORE': self.logger.debug(f"🚫 Ignoring file (category: IGNORE): {os.path.basename(file\_path)}") return False if file\_category == 'TEMPORARY': self.logger.debug(f"⏰ Temporary file, monitoring for context: {os.path.basename(file\_path)}") return True # Отслеживаем для контекста, но не создаем сессии if file\_category == 'MAIN': # Дополнительные проверки для основных файлов if not self.\_passes\_additional\_checks(file\_path): return False return True return False def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool: """Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)""" filename = os.path.basename(file\_path) # Используем ту же логику, но без проверки существования файла file\_category = self.get\_file\_category(file\_path) if file\_category == 'IGNORE': self.logger.debug(f"🚫 Ignoring deleted file (category: IGNORE): {filename}") return False if file\_category == 'TEMPORARY': self.logger.debug(f"⏰ Deleted temporary file: {filename}") return True if file\_category == 'MAIN': return True return False def \_is\_ignored\_file(self, filename: str) -> bool: """Проверяет является ли файл полностью игнорируемым""" filename\_lower = filename.lower() # Проверяем паттерны IGNORE for pattern in self.FILE\_CATEGORIES['IGNORE']: if self.\_matches\_pattern(filename\_lower, pattern): return True # Проверяем пользовательские ignore\_patterns for pattern in self.ignore\_patterns: if self.\_matches\_pattern(filename\_lower, pattern): return True return False def \_is\_temporary\_file(self, filename: str) -> bool: """Проверяет является ли файл временным - РАСШИРЕННАЯ ВЕРСИЯ""" filename\_lower = filename.lower() # Проверяем базовые паттерны TEMPORARY for pattern in self.FILE\_CATEGORIES['TEMPORARY']: if self.\_matches\_pattern(filename\_lower, pattern): return True # Проверяем расширенные паттерны для временных файлов for pattern in self.\_extended\_temp\_patterns: if re.match(pattern, filename, re.IGNORECASE): return True # Специфические паттерны для разных приложений specific\_temp\_patterns = [ r'~wrl\d+\.tmp', # Word r'~wrd\d+\.tmp', # Word r'~rf.\*\.tmp', # Excel r'.\*\.tmp\..\*', # Файлы с двойными расширениями r'^~\$.\*', # Автосохранение Office # Дополнительные паттерны для Excel/Word временных файлов r'^[A-F0-9]{8}\.tmp$', # E3327DC9.tmp и подобные r'^[A-F0-9]{8}$', # C1EE4200 и подобные (без расширения) ] for pattern in specific\_temp\_patterns: if re.match(pattern, filename\_lower): return True # Проверяем файлы без расширения с короткими именами (часто временные) name\_without\_ext = os.path.splitext(filename)[0] if '.' not in filename and len(name\_without\_ext) <= 8 and name\_without\_ext.isalnum(): if name\_without\_ext.isupper() or all(c in '0123456789ABCDEF' for c in name\_without\_ext.upper()): return True return False def \_matches\_pattern(self, filename: str, pattern: str) -> bool: """Проверяет совпадение с паттерном""" # Для паттернов, начинающихся с \* (например, "\*.tmp") if pattern.startswith('\*') and filename.endswith(pattern[1:]): return True # Для паттернов, заканчивающихся на \* (например, "~\*") elif pattern.endswith('\*') and filename.startswith(pattern[:-1]): return True # Для паттернов с \* в середине elif '\*' in pattern: if fnmatch.fnmatch(filename, pattern): return True # Точное совпадение elif filename == pattern: return True # Проверка по регулярному выражению elif self.\_is\_regex\_match(filename, pattern): return True return False def \_is\_regex\_match(self, filename: str, pattern: str) -> bool: """Проверяет совпадение с использованием упрощенных regex-паттернов""" try: # Преобразуем fnmatch-паттерн в regex regex\_pattern = fnmatch.translate(pattern) return re.match(regex\_pattern, filename) is not None except re.error: return False def \_passes\_additional\_checks(self, file\_path: str) -> bool: """Дополнительные проверки для основных файлов""" filename = os.path.basename(file\_path) # Проверяем игнорируемые расширения из конфига file\_ext = os.path.splitext(filename)[1].lower() if file\_ext in self.ignore\_extensions: self.logger.debug(f"Ignoring file due to extension: {filename}") return False # Проверяем игнорируемые директории в пути if self.\_contains\_ignore\_dirs(file\_path): self.logger.debug(f"Ignoring file in excluded directory: {file\_path}") return False # Проверяем размер файла (игнорируем слишком маленькие файлы) try: file\_size = os.path.getsize(file\_path) if file\_size < 10: # Игнорируем файлы меньше 10 байт self.logger.debug(f"Ignoring too small file: {filename} ({file\_size} bytes)") return False except (OSError, ValueError): pass return True def \_contains\_ignore\_dirs(self, file\_path: str) -> bool: """Проверяет содержит ли путь игнорируемые директории""" if not self.ignore\_dirs: return False # Нормализуем путь для кроссплатформенности normalized\_path = os.path.normpath(file\_path).lower() path\_parts = normalized\_path.split(os.sep) for ignore\_dir in self.ignore\_dirs: ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower() # Проверяем, содержится ли игнорируемая директория в пути if ignore\_dir\_normalized in path\_parts: return True return False def get\_monitorable\_files(self, directory: str) -> List[str]: """Возвращает список файлов для мониторинга в директории""" monitorable\_files = [] try: for root, dirs, files in os.walk(directory): # Исключаем игнорируемые директории из дальнейшего обхода dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))] for file in files: file\_path = os.path.join(root, file) if self.should\_monitor\_file(file\_path): monitorable\_files.append(file\_path) except Exception as e: self.logger.error(f"Error scanning directory {directory}: {e}") return monitorable\_files def \_should\_ignore\_dir(self, dir\_path: str) -> bool: """Определяет нужно ли игнорировать директорию""" dir\_name = os.path.basename(dir\_path) # Проверяем паттерны имен директорий for pattern in self.ignore\_patterns: if self.\_matches\_pattern(dir\_name, dir\_path): return True # Проверяем конкретные директории из ignore\_dirs if self.ignore\_dirs: normalized\_path = os.path.normpath(dir\_path).lower() for ignore\_dir in self.ignore\_dirs: ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower() # Проверяем точное совпадение имени директории if ignore\_dir\_normalized == dir\_name.lower(): return True # Проверяем вхождение в путь if ignore\_dir\_normalized in normalized\_path.split(os.sep): return True return False def clear\_cache(self): """Очищает кэш категорий""" self.\_category\_cache.clear()ow = time.time()

from typing import Dict, Optional, List

from shared.logger import setup\_logger

class SessionManager: uvicorn app.main:app --reload --host 127.0.0.1 --port 8000

uvicorn app.main:app --reload --host 127.0.0.1 --port 8000

uvicorn app.main:app --reload --host 127.0.0.1 --port 8000

uvicorn app.main:app --reload --host 127.0.0.1 --port 8000

uvicorn app.main:app --reload --host 127.0.0.1 --port 8000

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

self.active\_sessions: Dict[str, Dict] = {} # file\_path:username -> session\_data

self.closed\_sessions: Dict[str, List[Dict]] = {} # История закрытых сессий

self.logger = setup\_logger(\_\_name\_\_)

self.config = {}

def set\_config(self, config: dict):

"""Устанавливает конфигурацию"""

self.config = config

timeout = self.config.get('session\_timeout\_minutes', 1)

self.logger.info(f"⚙️ Session config: timeout={timeout}min, max\_age={self.config.get('max\_session\_hours', 3)}h")

def \_get\_session\_key(self, file\_path: str, username: str) -> str:

"""Генерирует ключ сессии"""

return f"{file\_path}:{username}"

def \_find\_recently\_closed(self, session\_key: str, hours: int = 1) -> Optional[Dict]:

"""Находит недавно закрытую сессию для возможного возобновления"""

if session\_key not in self.closed\_sessions:

return None

closed\_sessions = self.closed\_sessions[session\_key]

if not closed\_sessions:

return None

# Берем последнюю закрытую сессию

last\_session = closed\_sessions[-1]

# Проверяем, закрыта ли она в пределах указанного времени

if 'ended\_at' in last\_session and last\_session['ended\_at']:

time\_since\_close = datetime.now() - last\_session['ended\_at']

if time\_since\_close <= timedelta(hours=hours):

return last\_session

return None

def \_resume\_session(self, session\_data: Dict, file\_hash: str = None) -> Dict:

"""Возобновляет существующую сессию"""

session\_key = self.\_get\_session\_key(session\_data['file\_path'], session\_data['username'])

# Обновляем данные сессии

resumed\_session = session\_data.copy()

resumed\_session['last\_activity'] = datetime.now()

resumed\_session['resumed\_at'] = datetime.now()

resumed\_session['resume\_count'] = resumed\_session.get('resume\_count', 0) + 1

resumed\_session['hash\_before'] = file\_hash

# Убираем поля окончания, т.к. сессия снова активна

resumed\_session.pop('ended\_at', None)

resumed\_session.pop('hash\_after', None)

# Возвращаем в активные сессии

self.active\_sessions[session\_key] = resumed\_session

# Удаляем из истории закрытых, если она там есть

if session\_key in self.closed\_sessions and session\_data in self.closed\_sessions[session\_key]:

self.closed\_sessions[session\_key].remove(session\_data)

self.logger.info(f"🔄 Resumed session for {resumed\_session['file\_path']}")

return resumed\_session

def get\_active\_session(self, file\_path: str, username: str) -> Optional[Dict]:

"""Возвращает активную сессию для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.get(session\_key)

if session\_data:

# Проверяем не истекла ли сессия

if self.\_is\_session\_expired(session\_data):

# Сессия истекла - закрываем ее

self.logger.info(f"🕒 Session expired, closing: {file\_path}")

self.close\_session(file\_path, username)

return None

# Обновляем время последней активности

session\_data['last\_activity'] = datetime.now()

return session\_data

def \_is\_session\_expired(self, session\_data: Dict) -> bool:

"""Проверяет истекла ли сессия"""

timeout\_minutes = self.config.get('session\_timeout\_minutes', 1)

max\_age\_hours = self.config.get('max\_session\_hours', 3)

last\_activity = session\_data['last\_activity']

session\_age = datetime.now() - session\_data['started\_at']

time\_since\_activity = datetime.now() - last\_activity

timeout\_seconds = timeout\_minutes \* 60

# Проверяем таймаут активности

if time\_since\_activity.total\_seconds() > timeout\_seconds:

self.logger.info(f"🕒 Session expired by timeout: {session\_data['file\_path']}, inactive for {time\_since\_activity.total\_seconds():.1f}s > {timeout\_seconds}s")

return True

# Проверяем максимальный возраст сессии

if session\_age.total\_seconds() > (max\_age\_hours \* 3600):

self.logger.info(f"📅 Session expired by max age: {session\_data['file\_path']}, age: {session\_age.total\_seconds()/3600:.1f}h")

return True

return False

def check\_and\_close\_expired\_sessions(self) -> List[Dict]:

"""Проверяет и закрывает все просроченные сессии"""

expired\_sessions = []

total\_sessions = len(self.active\_sessions)

if total\_sessions == 0:

return expired\_sessions

self.logger.info(f"🔍 Checking {total\_sessions} active sessions for expiration...")

# Создаем копию списка для безопасной итерации

sessions\_to\_check = list(self.active\_sessions.items())

for session\_key, session\_data in sessions\_to\_check:

try:

if self.\_is\_session\_expired(session\_data):

file\_path = session\_data['file\_path']

username = session\_data['username']

time\_since\_activity = datetime.now() - session\_data['last\_activity']

self.logger.info(f"🔒 Closing expired session: {file\_path} (inactive: {time\_since\_activity.total\_seconds():.1f}s)")

# Закрываем сессию и получаем данные с ended\_at

closed\_data = self.close\_session(file\_path, username)

if closed\_data:

expired\_sessions.append(closed\_data)

self.logger.info(f"✅ Session closed with ended\_at: {closed\_data['ended\_at']}")

except Exception as e:

self.logger.error(f"❌ Error checking session {session\_key}: {e}")

if expired\_sessions:

self.logger.info(f"✅ Closed {len(expired\_sessions)} expired sessions")

else:

self.logger.debug(f"📊 All {total\_sessions} sessions are active")

return expired\_sessions

def smart\_create\_session(self, file\_path: str, username: str, file\_hash: str = None, resume\_window\_hours: int = 1) -> Dict:

"""Умное создание сессии с возможностью возобновления недавно закрытой сессии"""

# Сначала проверяем активную сессию

active\_session = self.get\_active\_session(file\_path, username)

if active\_session:

return active\_session

# Пытаемся найти недавно закрытую сессию для возобновления

session\_key = self.\_get\_session\_key(file\_path, username)

recently\_closed = self.\_find\_recently\_closed(session\_key, resume\_window\_hours)

if recently\_closed:

return self.\_resume\_session(recently\_closed, file\_hash)

else:

return self.create\_session(file\_path, username, file\_hash)

def create\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Создает новую сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = {

'session\_id': str(uuid.uuid4()),

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'username': username,

'started\_at': datetime.now(),

'last\_activity': datetime.now(),

'hash\_before': file\_hash,

'events': [],

'resume\_count': 0

}

self.active\_sessions[session\_key] = session\_data

self.logger.info(f"✅ Created session for {file\_path}")

return session\_data

def update\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Обновляет существующую сессию или создает новую"""

session\_data = self.get\_active\_session(file\_path, username)

if session\_data:

# Обновляем существующую сессию

if file\_hash:

session\_data['hash\_after'] = file\_hash

session\_data['last\_activity'] = datetime.now()

self.logger.debug(f"📝 Updated session for {file\_path}")

else:

# Создаем новую сессию

session\_data = self.create\_session(file\_path, username, file\_hash)

return session\_data

def close\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Закрывает сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.pop(session\_key, None)

if session\_data:

# Устанавливаем время окончания

ended\_at = datetime.now()

session\_data['ended\_at'] = ended\_at

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем в историю закрытых сессий

if session\_key not in self.closed\_sessions:

self.closed\_sessions[session\_key] = []

self.closed\_sessions[session\_key].append(session\_data)

# Ограничиваем историю

if len(self.closed\_sessions[session\_key]) > 10:

self.closed\_sessions[session\_key] = self.closed\_sessions[session\_key][-10:]

# Логируем с информацией о времени сессии

session\_duration = ended\_at - session\_data['started\_at']

self.logger.info(f"🔒 Closed session for {file\_path} (duration: {session\_duration.total\_seconds():.1f}s, ended\_at: {ended\_at})")

return session\_data

else:

self.logger.debug(f"❌ No active session found for: {file\_path} (user: {username})")

return None

def close\_all\_sessions\_for\_file(self, file\_path: str) -> List[Dict]:

"""Принудительно закрывает все сессии для указанного файла"""

closed\_sessions = []

sessions\_to\_close = []

for session\_key, session\_data in list(self.active\_sessions.items()):

if session\_data['file\_path'] == file\_path:

sessions\_to\_close.append((session\_data['file\_path'], session\_data['username']))

self.logger.info(f"🔍 Found {len(sessions\_to\_close)} sessions to close for: {file\_path}")

for file\_path, username in sessions\_to\_close:

session\_data = self.close\_session(file\_path, username)

if session\_data:

closed\_sessions.append(session\_data)

return closed\_sessions

def transfer\_session(self, old\_file\_path: str, new\_file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Переносит сессию со старого пути на новый"""

session\_key\_old = self.\_get\_session\_key(old\_file\_path, username)

session\_data = self.active\_sessions.pop(session\_key\_old, None)

if session\_data:

# Обновляем путь файла в данных сессии

session\_data['file\_path'] = new\_file\_path

session\_data['file\_name'] = os.path.basename(new\_file\_path)

session\_data['last\_activity'] = datetime.now()

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем с новым ключом

session\_key\_new = self.\_get\_session\_key(new\_file\_path, username)

self.active\_sessions[session\_key\_new] = session\_data

# Также обновляем в истории закрытых сессий, если нужно

if session\_key\_old in self.closed\_sessions:

self.closed\_sessions[session\_key\_new] = self.closed\_sessions.pop(session\_key\_old)

self.logger.info(f"🔄 Transferred session from {old\_file\_path} to {new\_file\_path}")

return session\_data

return None

def cleanup\_expired\_sessions(self, event\_handler) -> list:

"""Очищает просроченные сессии"""

return self.check\_and\_close\_expired\_sessions()# [file name]: session\_manager.py (УПРОЩЕННАЯ ВЕРСИЯ)

import os

import uuid

from datetime import datetime, timedelta

from typing import Dict, Optional, List

from shared.logger import setup\_logger

class SessionManager:

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

self.active\_sessions: Dict[str, Dict] = {} # file\_path:username -> session\_data

self.closed\_sessions: Dict[str, List[Dict]] = {} # История закрытых сессий

self.logger = setup\_logger(\_\_name\_\_)

self.config = {}

def set\_config(self, config: dict):

"""Устанавливает конфигурацию"""

self.config = config

timeout = self.config.get('session\_timeout\_minutes', 1)

self.logger.info(f"⚙️ Session config: timeout={timeout}min, max\_age={self.config.get('max\_session\_hours', 3)}h")

def \_get\_session\_key(self, file\_path: str, username: str) -> str:

"""Генерирует ключ сессии"""

return f"{file\_path}:{username}"

def \_find\_recently\_closed(self, session\_key: str, hours: int = 1) -> Optional[Dict]:

"""Находит недавно закрытую сессию для возможного возобновления"""

if session\_key not in self.closed\_sessions:

return None

closed\_sessions = self.closed\_sessions[session\_key]

if not closed\_sessions:

return None

# Берем последнюю закрытую сессию

last\_session = closed\_sessions[-1]

# Проверяем, закрыта ли она в пределах указанного времени

if 'ended\_at' in last\_session and last\_session['ended\_at']:

time\_since\_close = datetime.now() - last\_session['ended\_at']

if time\_since\_close <= timedelta(hours=hours):

return last\_session

return None

def \_resume\_session(self, session\_data: Dict, file\_hash: str = None) -> Dict:

"""Возобновляет существующую сессию"""

session\_key = self.\_get\_session\_key(session\_data['file\_path'], session\_data['username'])

# Обновляем данные сессии

resumed\_session = session\_data.copy()

resumed\_session['last\_activity'] = datetime.now()

resumed\_session['resumed\_at'] = datetime.now()

resumed\_session['resume\_count'] = resumed\_session.get('resume\_count', 0) + 1

resumed\_session['hash\_before'] = file\_hash

# Убираем поля окончания, т.к. сессия снова активна

resumed\_session.pop('ended\_at', None)

resumed\_session.pop('hash\_after', None)

# Возвращаем в активные сессии

self.active\_sessions[session\_key] = resumed\_session

# Удаляем из истории закрытых, если она там есть

if session\_key in self.closed\_sessions and session\_data in self.closed\_sessions[session\_key]:

self.closed\_sessions[session\_key].remove(session\_data)

self.logger.info(f"🔄 Resumed session for {resumed\_session['file\_path']}")

return resumed\_session

def get\_active\_session(self, file\_path: str, username: str) -> Optional[Dict]:

"""Возвращает активную сессию для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.get(session\_key)

if session\_data:

# Проверяем не истекла ли сессия

if self.\_is\_session\_expired(session\_data):

# Сессия истекла - закрываем ее

self.logger.info(f"🕒 Session expired, closing: {file\_path}")

self.close\_session(file\_path, username)

return None

# Обновляем время последней активности

session\_data['last\_activity'] = datetime.now()

return session\_data

def \_is\_session\_expired(self, session\_data: Dict) -> bool:

"""Проверяет истекла ли сессия"""

timeout\_minutes = self.config.get('session\_timeout\_minutes', 1)

max\_age\_hours = self.config.get('max\_session\_hours', 3)

last\_activity = session\_data['last\_activity']

session\_age = datetime.now() - session\_data['started\_at']

time\_since\_activity = datetime.now() - last\_activity

timeout\_seconds = timeout\_minutes \* 60

# Проверяем таймаут активности

if time\_since\_activity.total\_seconds() > timeout\_seconds:

self.logger.info(f"🕒 Session expired by timeout: {session\_data['file\_path']}, inactive for {time\_since\_activity.total\_seconds():.1f}s > {timeout\_seconds}s")

return True

# Проверяем максимальный возраст сессии

if session\_age.total\_seconds() > (max\_age\_hours \* 3600):

self.logger.info(f"📅 Session expired by max age: {session\_data['file\_path']}, age: {session\_age.total\_seconds()/3600:.1f}h")

return True

return False

def check\_and\_close\_expired\_sessions(self) -> List[Dict]:

"""Проверяет и закрывает все просроченные сессии"""

expired\_sessions = []

total\_sessions = len(self.active\_sessions)

if total\_sessions == 0:

return expired\_sessions

self.logger.info(f"🔍 Checking {total\_sessions} active sessions for expiration...")

# Создаем копию списка для безопасной итерации

sessions\_to\_check = list(self.active\_sessions.items())

for session\_key, session\_data in sessions\_to\_check:

try:

if self.\_is\_session\_expired(session\_data):

file\_path = session\_data['file\_path']

username = session\_data['username']

time\_since\_activity = datetime.now() - session\_data['last\_activity']

self.logger.info(f"🔒 Closing expired session: {file\_path} (inactive: {time\_since\_activity.total\_seconds():.1f}s)")

# Закрываем сессию и получаем данные с ended\_at

closed\_data = self.close\_session(file\_path, username)

if closed\_data:

expired\_sessions.append(closed\_data)

self.logger.info(f"✅ Session closed with ended\_at: {closed\_data['ended\_at']}")

except Exception as e:

self.logger.error(f"❌ Error checking session {session\_key}: {e}")

if expired\_sessions:

self.logger.info(f"✅ Closed {len(expired\_sessions)} expired sessions")

else:

self.logger.debug(f"📊 All {total\_sessions} sessions are active")

return expired\_sessions

def smart\_create\_session(self, file\_path: str, username: str, file\_hash: str = None, resume\_window\_hours: int = 1) -> Dict:

"""Умное создание сессии с возможностью возобновления недавно закрытой сессии"""

# Сначала проверяем активную сессию

active\_session = self.get\_active\_session(file\_path, username)

if active\_session:

return active\_session

# Пытаемся найти недавно закрытую сессию для возобновления

session\_key = self.\_get\_session\_key(file\_path, username)

recently\_closed = self.\_find\_recently\_closed(session\_key, resume\_window\_hours)

if recently\_closed:

return self.\_resume\_session(recently\_closed, file\_hash)

else:

return self.create\_session(file\_path, username, file\_hash)

def create\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Создает новую сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = {

'session\_id': str(uuid.uuid4()),

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'username': username,

'started\_at': datetime.now(),

'last\_activity': datetime.now(),

'hash\_before': file\_hash,

'events': [],

'resume\_count': 0

}

self.active\_sessions[session\_key] = session\_data

self.logger.info(f"✅ Created session for {file\_path}")

return session\_data

def update\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Обновляет существующую сессию или создает новую"""

session\_data = self.get\_active\_session(file\_path, username)

if session\_data:

# Обновляем существующую сессию

if file\_hash:

session\_data['hash\_after'] = file\_hash

session\_data['last\_activity'] = datetime.now()

self.logger.debug(f"📝 Updated session for {file\_path}")

else:

# Создаем новую сессию

session\_data = self.create\_session(file\_path, username, file\_hash)

return session\_data

def close\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Закрывает сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.pop(session\_key, None)

if session\_data:

# Устанавливаем время окончания

ended\_at = datetime.now()

session\_data['ended\_at'] = ended\_at

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем в историю закрытых сессий

if session\_key not in self.closed\_sessions:

self.closed\_sessions[session\_key] = []

self.closed\_sessions[session\_key].append(session\_data)

# Ограничиваем историю

if len(self.closed\_sessions[session\_key]) > 10:

self.closed\_sessions[session\_key] = self.closed\_sessions[session\_key][-10:]

# Логируем с информацией о времени сессии

session\_duration = ended\_at - session\_data['started\_at']

self.logger.info(f"🔒 Closed session for {file\_path} (duration: {session\_duration.total\_seconds():.1f}s, ended\_at: {ended\_at})")

return session\_data

else:

self.logger.debug(f"❌ No active session found for: {file\_path} (user: {username})")

return None

def close\_all\_sessions\_for\_file(self, file\_path: str) -> List[Dict]:

"""Принудительно закрывает все сессии для указанного файла"""

closed\_sessions = []

sessions\_to\_close = []

for session\_key, session\_data in list(self.active\_sessions.items()):

if session\_data['file\_path'] == file\_path:

sessions\_to\_close.append((session\_data['file\_path'], session\_data['username']))

self.logger.info(f"🔍 Found {len(sessions\_to\_close)} sessions to close for: {file\_path}")

for file\_path, username in sessions\_to\_close:

session\_data = self.close\_session(file\_path, username)

if session\_data:

closed\_sessions.append(session\_data)

return closed\_sessions

def transfer\_session(self, old\_file\_path: str, new\_file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Переносит сессию со старого пути на новый"""

session\_key\_old = self.\_get\_session\_key(old\_file\_path, username)

session\_data = self.active\_sessions.pop(session\_key\_old, None)

if session\_data:

# Обновляем путь файла в данных сессии

session\_data['file\_path'] = new\_file\_path

session\_data['file\_name'] = os.path.basename(new\_file\_path)

session\_data['last\_activity'] = datetime.now()

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем с новым ключом

session\_key\_new = self.\_get\_session\_key(new\_file\_path, username)

self.active\_sessions[session\_key\_new] = session\_data

# Также обновляем в истории закрытых сессий, если нужно

if session\_key\_old in self.closed\_sessions:

self.closed\_sessions[session\_key\_new] = self.closed\_sessions.pop(session\_key\_old)

self.logger.info(f"🔄 Transferred session from {old\_file\_path} to {new\_file\_path}")

return session\_data

return None

def cleanup\_expired\_sessions(self, event\_handler) -> list:

"""Очищает просроченные сессии"""

return self.check\_and\_close\_expired\_sessions()

def get\_session\_stats(self) -> Dict:

"""Возвращает статистику по сессиям"""

total\_resumes = sum(session.get('resume\_count', 0) for session in self.active\_sessions.values())

return {

'active\_sessions': len(self.active\_sessions),

'session\_keys': list(self.active\_sessions.keys()),

'total\_resumes': total\_resumes,

'closed\_sessions\_count': sum(len(sessions) for sessions in self.closed\_sessions.values()),

}

def get\_session\_history(self, file\_path: str, username: str) -> List[Dict]:

"""Возвращает историю сессий для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

return self.closed\_sessions.get(session\_key, [])

def get\_session\_stats(self) -> Dict:

"""Возвращает статистику по сессиям"""

total\_resumes = sum(session.get('resume\_count', 0) for session in self.active\_sessions.values())

return {

'active\_sessions': len(self.active\_sessions),

'session\_keys': list(self.active\_sessions.keys()),

'total\_resumes': total\_resumes,

'closed\_sessions\_count': sum(len(sessions) for sessions in self.closed\_sessions.values()),

}

def get\_session\_history(self, file\_path: str, username: str) -> List[Dict]:

"""Возвращает историю сессий для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

return self.closed\_sessions.get(session\_key, [])g deleted file in excluded directory: {file\_path}")

return False

import os

import uuid

from datetime import datetimeimporimport os

import time

import threading

from datetime import datetime

from shared.logger import setup\_logger

from shared.config\_loader import get\_monitoring\_config

from .event\_handler import EventHandler

from .background\_checker import BackgroundSessionChecker

class FileMonitor:

def \_\_init\_\_(self, monitoring\_config=None):

if monitoring\_config is None:

self.config = get\_monitoring\_config()

else:

self.config = monitoring\_config

self.logger = setup\_logger(\_\_name\_\_)

# Инициализация компонентов

self.event\_handler = EventHandler(monitoring\_config=self.config)

# Создаем фоновый проверщик сессий

check\_interval = self.config.get('background\_check\_interval', 15)

self.background\_checker = BackgroundSessionChecker(

self.event\_handler,

check\_interval=check\_interval

)

# Флаг работы

self.\_running = False

self.\_monitor\_thread = None

# Настройки сканирования

self.scan\_interval = self.config.get('poll\_interval', 1)

self.watch\_paths = self.config.get('watch\_paths', ['./monitor'])

# Трекер состояния файлов

self.file\_states = {} # file\_path -> (mtime, size)

self.logger.info(f"🎯 FileMonitor initialized with process-based scanning every {self.scan\_interval}s")

def start(self):

"""Запускает мониторинг"""

if self.\_running:

self.logger.warning("Monitor is already running")

return

self.\_running = True

# Проверяем соединение с API

if not self.event\_handler.api\_client.test\_connection():

self.logger.error("❌ Cannot connect to API server")

return

# Запускаем фоновую проверку

self.background\_checker.start()

# Запускаем основной цикл мониторинга

self.\_monitor\_thread = threading.Thread(target=self.\_monitor\_loop, daemon=True)

self.\_monitor\_thread.start()

self.logger.info("✅ File monitoring started successfully")

# Основной цикл (только для статистики)

try:

stats\_interval = 30

while self.\_running:

time.sleep(stats\_interval)

# Логируем статистику

stats = self.event\_handler.get\_stats()

active\_sessions = stats['active\_sessions']

self.logger.info(f"📊 Stats: {active\_sessions} active sessions, {stats.get('expired\_sessions', 0)} expired")

# Детальная информация об активных сессиях

if active\_sessions > 0:

timeout\_minutes = self.event\_handler.session\_manager.config.get('session\_timeout\_minutes', 30)

for session\_key, session\_data in self.event\_handler.session\_manager.active\_sessions.items():

last\_activity = session\_data['last\_activity']

time\_since\_activity = datetime.now() - last\_activity

remaining = (timeout\_minutes \* 60) - time\_since\_activity.total\_seconds()

self.logger.debug(f"⏰ {session\_key}: inactive {time\_since\_activity.total\_seconds():.1f}s (expires in {remaining:.1f}s)")

except KeyboardInterrupt:

self.stop()

except Exception as e:

self.logger.error(f"❌ Unexpected error: {e}")

self.stop()

def \_monitor\_loop(self):

"""Основной цикл мониторинга файлов"""

self.logger.info("🔄 Starting file monitoring loop...")

# Первоначальное сканирование для отслеживания состояния

self.\_initial\_scan()

while self.\_running:

try:

self.\_scan\_files()

time.sleep(self.scan\_interval)

except Exception as e:

self.logger.error(f"Error in monitor loop: {e}")

time.sleep(5) # Пауза при ошибке

def \_initial\_scan(self):

"""Первоначальное сканирование файлов для установки базового состояния"""

self.logger.info("🔍 Performing initial file scan for state tracking...")

for watch\_path in self.watch\_paths:

if not os.path.exists(watch\_path):

self.logger.warning(f"Watch path does not exist: {watch\_path}")

continue

for root, dirs, files in os.walk(watch\_path):

# Пропускаем игнорируемые директории

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.event\_handler.file\_validator.should\_monitor\_file(file\_path):

try:

stat = os.stat(file\_path)

self.file\_states[file\_path] = (stat.st\_mtime, stat.st\_size)

self.logger.debug(f"📁 Tracked existing file: {file\_path}")

except (OSError, PermissionError) as e:

self.logger.debug(f"Could not access file {file\_path}: {e}")

def \_scan\_files(self):

"""Сканирует файлы на изменения"""

current\_files = set()

for watch\_path in self.watch\_paths:

if not os.path.exists(watch\_path):

continue

for root, dirs, files in os.walk(watch\_path):

# Пропускаем игнорируемые директории

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if not self.event\_handler.file\_validator.should\_monitor\_file(file\_path):

continue

current\_files.add(file\_path)

self.\_check\_file\_changes(file\_path)

# Проверяем удаленные файлы

self.\_check\_deleted\_files(current\_files)

def \_check\_file\_changes(self, file\_path):

"""Проверяет изменения файла"""

try:

if not os.path.exists(file\_path):

return

current\_stat = os.stat(file\_path)

current\_mtime = current\_stat.st\_mtime

current\_size = current\_stat.st\_size

previous\_state = self.file\_states.get(file\_path)

if previous\_state is None:

# Новый файл

self.logger.info(f"🆕 New file detected: {file\_path}")

self.\_process\_file\_event(file\_path, 'created', current\_mtime, current\_size)

else:

prev\_mtime, prev\_size = previous\_state

if current\_mtime != prev\_mtime:

if current\_size != prev\_size:

# Файл изменен

self.logger.debug(f"📝 File modified: {file\_path}")

self.\_process\_file\_event(file\_path, 'modified', current\_mtime, current\_size)

else:

# Изменены только метаданные

self.file\_states[file\_path] = (current\_mtime, current\_size)

except (OSError, PermissionError) as e:

self.logger.debug(f"Could not access file {file\_path}: {e}")

def \_check\_deleted\_files(self, current\_files):

"""Проверяет удаленные файлы"""

deleted\_files = set(self.file\_states.keys()) - current\_files

for file\_path in deleted\_files:

if self.event\_handler.file\_validator.should\_monitor\_file\_by\_name(file\_path):

self.logger.info(f"🗑️ File deleted: {file\_path}")

self.\_process\_file\_event(file\_path, 'deleted')

# Удаляем из состояния в любом случае

if file\_path in self.file\_states:

del self.file\_states[file\_path]

def \_process\_file\_event(self, file\_path, event\_type, mtime=None, size=None):

"""Обрабатывает событие файла"""

try:

# Обрабатываем событие через event\_handler

success = self.event\_handler.handle\_file\_event(event\_type, file\_path)

if success and event\_type != 'deleted':

# Обновляем состояние только для существующих файлов

if mtime is None or size is None:

stat = os.stat(file\_path)

mtime, size = stat.st\_mtime, stat.st\_size

self.file\_states[file\_path] = (mtime, size)

except Exception as e:

self.logger.error(f"Error processing {event\_type} event for {file\_path}: {e}")

def \_should\_ignore\_dir(self, dir\_path):

"""Проверяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

ignore\_dirs = self.config.get('ignore\_dirs', [])

if dir\_name in ignore\_dirs:

return True

# Проверяем паттерны

ignore\_patterns = self.config.get('ignore\_patterns', [])

for pattern in ignore\_patterns:

if pattern.startswith('\*') and dir\_name.endswith(pattern[1:]):

return True

elif pattern.endswith('\*') and dir\_name.startswith(pattern[:-1]):

return True

elif pattern in dir\_name:

return True

return False

def stop(self):

"""Останавливает мониторинг"""

self.logger.info("🛑 Stopping file monitor...")

self.\_running = False

if self.background\_checker:

self.background\_checker.stop()

if self.\_monitor\_thread and self.\_monitor\_thread.is\_alive():

self.\_monitor\_thread.join(timeout=5)

# Очищаем ресурсы

self.event\_handler.cleanup()

self.logger.info("✅ File monitoring stopped")t os

import getpass

import platform

from datetime import datetime, timedelta

from typing import Dict, Any, Optional

from shared.logger import setup\_logger

from shared.config\_loader import get\_monitoring\_config, get\_api\_client\_config

try:

import win32security # For Windows user

except ImportError:

win32security = None

try:

import psutil # For tracking open files

except ImportError:

psutil = None

from .hash\_calculator import HashCalculator

from .session\_manager import SessionManager

from .api\_client import APIClient

from .file\_validator import FileValidator

class EventHandler:

def \_\_init\_\_(self, monitoring\_config=None):

if monitoring\_config is None:

self.config = get\_monitoring\_config()

else:

self.config = monitoring\_config

self.logger = setup\_logger(\_\_name\_\_)

# Инициализация компонентов

self.hash\_calculator = HashCalculator(self.config.get('hashing', {}))

self.session\_manager = SessionManager()

# ПЕРЕДАЕМ КОНФИГУРАЦИЮ СЕССИЙ ПРАВИЛЬНО

session\_config = self.config.get('sessions', {})

self.session\_manager.set\_config(session\_config)

self.api\_client = APIClient()

self.file\_validator = FileValidator(self.config)

# Статистика

self.stats = {

'events\_processed': 0,

'events\_failed': 0,

'sessions\_created': 0,

'sessions\_resumed': 0,

'files\_closed': 0,

'files\_deleted': 0,

'expired\_sessions': 0,

'main\_files\_processed': 0,

'temporary\_files\_ignored': 0

}

# Отслеживание открытых файлов

self.open\_files = {} # file\_path -> {username, processes, last\_activity}

# Трекер перемещений файлов - УЛУЧШЕННАЯ ВЕРСИЯ

self.file\_renames = {} # src\_path -> dest\_path

self.file\_move\_chains = {} # track complex move operations

self.temp\_to\_main\_map = {} # temp\_file -> main\_file (для Office операций)

self.main\_file\_tracking = {} # main\_file -> последнее известное состояние

# Время последней проверки открытых файлов

self.last\_open\_files\_check = datetime.now()

# ДОБАВЛЕНО: Фильтр массовых событий

self.recent\_events = {} # file\_path -> last\_event\_time

self.event\_cooldown = 2.0 # секунды между событиями для одного файла

# ДОБАВЛЕНО: Трекер реально открытых файлов

self.verified\_open\_files = set() # файлы, которые реально открыты в процессах

self.logger.info("EventHandler initialized with enhanced file classification strategy")

def handle\_file\_event(self, event\_type: str, file\_path: str, dest\_path: str = None) -> bool:

"""Обрабатывает событие файла - УЛУЧШЕННАЯ ВЕРСИЯ"""

try:

# ДОБАВЛЕНО: Фильтр частых событий

if not self.\_should\_process\_event(file\_path, event\_type):

self.logger.debug(f"⏰ Skipping frequent event: {event\_type} for {file\_path}")

return True

self.stats['events\_processed'] += 1

self.logger.debug(f"Raw event: {event\_type} - {file\_path} -> {dest\_path}")

# Определяем категорию файла ДО обработки

file\_category = self.file\_validator.get\_file\_category(file\_path)

# Обрабатываем перемещение как специальный случай

if event\_type == 'moved' and dest\_path:

return self.\_handle\_file\_moved(file\_path, dest\_path, file\_category)

# Для IGNORE файлов - полностью пропускаем обработку

if file\_category == 'IGNORE':

self.logger.debug(f"🚫 Completely ignoring event for IGNORE file: {file\_path}")

return True

# Для TEMPORARY файлов - обрабатываем для контекста, но не создаем сессии

if file\_category == 'TEMPORARY':

self.stats['temporary\_files\_ignored'] += 1

self.logger.debug(f"⏰ Processing temporary file for context: {file\_path}")

return self.\_handle\_temporary\_file(event\_type, file\_path, dest\_path)

# Для MAIN файлов - полная обработка с сессиями

if file\_category == 'MAIN':

self.stats['main\_files\_processed'] += 1

return self.\_handle\_main\_file(event\_type, file\_path)

self.logger.warning(f"Unknown file category for {file\_path}: {file\_category}")

return False

except Exception as e:

self.stats['events\_failed'] += 1

self.logger.error(f"Error handling {event\_type} event for {file\_path}: {e}")

return False

def \_handle\_main\_file(self, event\_type: str, file\_path: str) -> bool:

"""Обрабатывает событие для основного файла (с созданием сессий)"""

# Проверяем нужно ли отслеживать файл (дополнительные проверки)

if not self.file\_validator.should\_monitor\_file(file\_path):

self.logger.debug(f"Ignoring main file: {file\_path}")

return True

# Получаем пользователя ОС и НОРМАЛИЗУЕМ его имя

username = self.\_get\_file\_modifier\_safe(file\_path, event\_type)

normalized\_username = self.\_normalize\_username(username)

self.logger.debug(f"Main file event: {event\_type} - {file\_path} by {normalized\_username}")

# ДОБАВЛЕНО: Проверяем реальное открытие файла перед созданием сессии

if event\_type == 'modified':

if not self.\_is\_file\_really\_opened(file\_path):

self.logger.debug(f"📁 Main file not actually opened, skipping session: {file\_path}")

return True

# Обновляем отслеживание открытых файлов

if event\_type in ('created', 'modified'):

self.\_update\_open\_file\_tracking(file\_path, normalized\_username, event\_type)

elif event\_type == 'deleted':

# Удаляем из отслеживания открытых файлов

if file\_path in self.open\_files:

del self.open\_files[file\_path]

# Вычисляем хеш если нужно (только для существующих файлов)

file\_hash = None

if (event\_type != 'deleted' and

self.config.get('hashing', {}).get('enabled', True) and

os.path.exists(file\_path)):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Обрабатываем в зависимости от типа события

if event\_type == 'created':

return self.\_handle\_file\_created(file\_path, normalized\_username, file\_hash)

elif event\_type == 'modified':

return self.\_handle\_file\_modified(file\_path, normalized\_username, file\_hash)

elif event\_type == 'deleted':

return self.\_handle\_file\_deleted(file\_path, normalized\_username)

else:

self.logger.warning(f"Unknown event type for main file: {event\_type}")

return False

def \_handle\_temporary\_file(self, event\_type: str, file\_path: str, dest\_path: str = None) -> bool:

"""Обрабатывает событие для временного файла (без создания сессий)"""

# Для временных файлов просто логируем и отслеживаем для контекста

self.logger.debug(f"Temporary file event: {event\_type} - {file\_path}")

# Если это перемещение временного файла, сохраняем информацию для контекста

if event\_type == 'moved' and dest\_path:

dest\_category = self.file\_validator.get\_file\_category(dest\_path)

self.logger.debug(f"Temporary file moved: {file\_path} -> {dest\_path} (dest category: {dest\_category})")

# Если временный файл перемещается в основной - это может быть операцией сохранения

if dest\_category == 'MAIN':

self.logger.info(f"🔄 Temporary -> Main file operation detected: {file\_path} -> {dest\_path}")

# Сохраняем связь для возможного восстановления сессии

self.temp\_to\_main\_map[file\_path] = dest\_path

return True

def \_handle\_file\_moved(self, src\_path: str, dest\_path: str, src\_category: str) -> bool:

"""Обрабатывает перемещение/переименование файла - УЛУЧШЕННАЯ ВЕРСИЯ"""

self.logger.info(f"🔄 File moved: {src\_path} -> {dest\_path} (src category: {src\_category})")

dest\_category = self.file\_validator.get\_file\_category(dest\_path)

self.logger.debug(f"Destination category: {dest\_category}")

# Определяем тип операции

operation\_type = self.\_classify\_move\_operation(src\_path, dest\_path, src\_category, dest\_category)

self.logger.debug(f"Move operation type: {operation\_type}")

if operation\_type == 'TEMP\_TO\_TEMP':

# Временный -> временный - просто отслеживаем цепочку

self.file\_move\_chains[src\_path] = dest\_path

self.logger.debug(f"Temp-to-temp move: {src\_path} -> {dest\_path}")

return True

elif operation\_type == 'MAIN\_TO\_TEMP':

# Основной -> временный - сохраняем связь

self.temp\_to\_main\_map[dest\_path] = src\_path

self.logger.debug(f"Main-to-temp move: {src\_path} -> {dest\_path}")

# Сохраняем информацию об основном файле

self.main\_file\_tracking[src\_path] = {

'last\_seen': datetime.now(),

'temp\_file': dest\_path

}

return True

elif operation\_type == 'TEMP\_TO\_MAIN':

# Временный -> основной - восстанавливаем сессию

return self.\_handle\_temp\_to\_main\_move(src\_path, dest\_path)

elif operation\_type == 'MAIN\_TO\_MAIN':

# Основной -> основной - переносим сессию

return self.\_handle\_main\_to\_main\_move(src\_path, dest\_path)

elif operation\_type == 'TEMP\_TO\_IGNORE':

# Временный -> игнорируемый - обычная операция Excel

self.logger.debug(f"Temp-to-ignore move (Excel operation): {src\_path} -> {dest\_path}")

return True

elif operation\_type == 'IGNORE\_TO\_MAIN':

# Игнорируемый -> основной - восстанавливаем сессию

return self.\_handle\_ignore\_to\_main\_move(src\_path, dest\_path)

else:

self.logger.warning(f"Unknown move operation type: {operation\_type}")

# Для неизвестных операций пробуем определить по контексту

return self.\_handle\_unknown\_move\_operation(src\_path, dest\_path, src\_category, dest\_category)

def \_classify\_move\_operation(self, src\_path: str, dest\_path: str, src\_category: str, dest\_category: str) -> str:

"""Классифицирует тип операции перемещения"""

if src\_category == 'TEMPORARY' and dest\_category == 'TEMPORARY':

return 'TEMP\_TO\_TEMP'

elif src\_category == 'MAIN' and dest\_category == 'TEMPORARY':

return 'MAIN\_TO\_TEMP'

elif src\_category == 'TEMPORARY' and dest\_category == 'MAIN':

return 'TEMP\_TO\_MAIN'

elif src\_category == 'MAIN' and dest\_category == 'MAIN':

return 'MAIN\_TO\_MAIN'

elif src\_category == 'TEMPORARY' and dest\_category == 'IGNORE':

return 'TEMP\_TO\_IGNORE'

elif src\_category == 'IGNORE' and dest\_category == 'MAIN':

return 'IGNORE\_TO\_MAIN'

elif src\_category == 'IGNORE' and dest\_category == 'TEMPORARY':

return 'IGNORE\_TO\_TEMP'

else:

return 'UNKNOWN'

def \_handle\_temp\_to\_main\_move(self, src\_path: str, dest\_path: str) -> bool:

"""Обрабатывает перемещение временного файла в основной"""

# Ищем основной файл в маппинге

main\_file = self.temp\_to\_main\_map.get(src\_path)

username = self.\_get\_file\_modifier\_safe(dest\_path, 'moved')

normalized\_username = self.\_normalize\_username(username)

file\_hash = None

if self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(dest\_path)

if main\_file:

# Нашли связь - переносим сессию с основного файла

old\_session = self.session\_manager.get\_active\_session(main\_file, normalized\_username)

if old\_session:

transferred\_session = self.session\_manager.transfer\_session(

main\_file, dest\_path, normalized\_username, file\_hash

)

if transferred\_session:

self.logger.info(f"🔄 Transferred session from main file {main\_file} to {dest\_path}")

# Очищаем маппинг

if src\_path in self.temp\_to\_main\_map:

del self.temp\_to\_main\_map[src\_path]

if main\_file in self.main\_file\_tracking:

del self.main\_file\_tracking[main\_file]

# Отправляем событие moved

return self.\_send\_moved\_event(src\_path, dest\_path, normalized\_username, file\_hash)

# Если не нашли связь, создаем новую сессию

self.session\_manager.smart\_create\_session(dest\_path, normalized\_username, file\_hash)

self.logger.info(f"✅ Created new session for moved file {dest\_path}")

# Очищаем маппинг на всякий случай

if src\_path in self.temp\_to\_main\_map:

del self.temp\_to\_main\_map[src\_path]

return self.\_send\_moved\_event(src\_path, dest\_path, normalized\_username, file\_hash)

def \_handle\_main\_to\_main\_move(self, src\_path: str, dest\_path: str) -> bool:

"""Обрабатывает перемещение основного файла в основной"""

username = self.\_get\_file\_modifier\_safe(dest\_path, 'moved')

normalized\_username = self.\_normalize\_username(username)

file\_hash = None

if self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(dest\_path)

# Пытаемся перенести существующую сессию

session\_transferred = False

old\_session = self.session\_manager.get\_active\_session(src\_path, normalized\_username)

if old\_session:

transferred\_session = self.session\_manager.transfer\_session(

src\_path, dest\_path, normalized\_username, file\_hash

)

if transferred\_session:

session\_transferred = True

self.logger.info(f"✅ Transferred session from {src\_path} to {dest\_path}")

# Если не нашли существующую сессию, создаем новую

if not session\_transferred:

self.session\_manager.smart\_create\_session(dest\_path, normalized\_username, file\_hash)

self.logger.info(f"✅ Created new session for moved file {dest\_path}")

# Сохраняем информацию о перемещении

self.file\_renames[src\_path] = dest\_path

self.file\_move\_chains[src\_path] = dest\_path

return self.\_send\_moved\_event(src\_path, dest\_path, normalized\_username, file\_hash)

def \_handle\_ignore\_to\_main\_move(self, src\_path: str, dest\_path: str) -> bool:

"""Обрабатывает перемещение игнорируемого файла в основной (типично для Excel)"""

username = self.\_get\_file\_modifier\_safe(dest\_path, 'moved')

normalized\_username = self.\_normalize\_username(username)

file\_hash = None

if self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(dest\_path)

# Ищем связанный основной файл по имени или пути

main\_file = self.\_find\_related\_main\_file(src\_path, dest\_path)

if main\_file:

# Нашли связанный файл - переносим сессию

old\_session = self.session\_manager.get\_active\_session(main\_file, normalized\_username)

if old\_session:

transferred\_session = self.session\_manager.transfer\_session(

main\_file, dest\_path, normalized\_username, file\_hash

)

if transferred\_session:

self.logger.info(f"🔄 Transferred session from related file {main\_file} to {dest\_path}")

return self.\_send\_moved\_event(src\_path, dest\_path, normalized\_username, file\_hash)

# Создаем новую сессию для основного файла

self.session\_manager.smart\_create\_session(dest\_path, normalized\_username, file\_hash)

self.logger.info(f"✅ Created new session for Excel file {dest\_path}")

return self.\_send\_moved\_event(src\_path, dest\_path, normalized\_username, file\_hash)

def \_handle\_unknown\_move\_operation(self, src\_path: str, dest\_path: str, src\_category: str, dest\_category: str) -> bool:

"""Обрабатывает неизвестные операции перемещения на основе эвристик"""

username = self.\_get\_file\_modifier\_safe(dest\_path, 'moved')

normalized\_username = self.\_normalize\_username(username)

file\_hash = None

if self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(dest\_path)

# Эвристика: если конечный файл основной - создаем/переносим сессию

if dest\_category == 'MAIN':

# Пытаемся найти связанный файл для переноса сессии

related\_file = self.\_find\_related\_main\_file(src\_path, dest\_path)

if related\_file:

old\_session = self.session\_manager.get\_active\_session(related\_file, normalized\_username)

if old\_session:

self.session\_manager.transfer\_session(related\_file, dest\_path, normalized\_username, file\_hash)

self.logger.info(f"🔄 Transferred session from related file {related\_file} to {dest\_path}")

else:

self.session\_manager.smart\_create\_session(dest\_path, normalized\_username, file\_hash)

self.logger.info(f"✅ Created new session for {dest\_path}")

else:

self.session\_manager.smart\_create\_session(dest\_path, normalized\_username, file\_hash)

self.logger.info(f"✅ Created new session for {dest\_path}")

# Всегда отправляем событие moved

return self.\_send\_moved\_event(src\_path, dest\_path, normalized\_username, file\_hash)

def \_find\_related\_main\_file(self, src\_path: str, dest\_path: str) -> Optional[str]:

"""Находит связанный основной файл по имени или пути"""

src\_name = os.path.basename(src\_path)

dest\_name = os.path.basename(dest\_path)

# Ищем в цепочках перемещений

for chain\_src, chain\_dest in self.file\_move\_chains.items():

if chain\_dest == src\_path:

chain\_category = self.file\_validator.get\_file\_category(chain\_src)

if chain\_category == 'MAIN':

return chain\_src

# Ищем в маппинге временных файлов

for temp\_file, main\_file in self.temp\_to\_main\_map.items():

if temp\_file == src\_path:

return main\_file

# Ищем по схожести имен (для Excel операций)

if src\_name.isalnum() and len(src\_name) == 8: # Типичный временный файл Excel

# Ищем основной файл в той же директории

dir\_path = os.path.dirname(dest\_path)

for known\_main in self.main\_file\_tracking.keys():

if os.path.dirname(known\_main) == dir\_path:

return known\_main

return None

def \_send\_moved\_event(self, src\_path: str, dest\_path: str, username: str, file\_hash: str) -> bool:

"""Отправляет событие перемещения"""

event\_data = {

'file\_path': dest\_path,

'file\_name': os.path.basename(dest\_path),

'old\_file\_path': src\_path,

'old\_file\_name': os.path.basename(src\_path),

'event\_type': 'moved',

'file\_hash': file\_hash,

'user\_id': username,

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"Failed to send moved event for {src\_path} -> {dest\_path}")

return success

# Остальные методы остаются без изменений

def \_normalize\_username(self, username: str) -> str:

"""Нормализует имя пользователя к единому формату"""

if not username:

return getpass.getuser()

if '\\' in username:

parts = username.split('\\')

normalized = parts[-1]

self.logger.debug(f"Normalized username: {username} -> {normalized}")

return normalized

return username

def \_should\_process\_event(self, file\_path: str, event\_type: str) -> bool:

"""Определяет нужно ли обрабатывать событие (фильтр частых событий)"""

current\_time = datetime.now()

event\_key = f"{file\_path}:{event\_type}"

if event\_type in ('deleted', 'moved'):

return True

if event\_key in self.recent\_events:

time\_since\_last = (current\_time - self.recent\_events[event\_key]).total\_seconds()

if time\_since\_last < self.event\_cooldown:

return False

self.recent\_events[event\_key] = current\_time

# Очищаем старые записи (старше 10 секунд)

old\_entries = []

for key, event\_time in self.recent\_events.items():

if (current\_time - event\_time).total\_seconds() > 10:

old\_entries.append(key)

for key in old\_entries:

del self.recent\_events[key]

return True

def \_is\_file\_really\_opened(self, file\_path: str) -> bool:

"""Проверяет, действительно ли файл открыт в каком-либо процессе"""

if not psutil:

return True

try:

processes = self.\_get\_processes\_using\_file(file\_path)

is\_opened = len(processes) > 0

if is\_opened:

self.verified\_open\_files.add(file\_path)

self.logger.debug(f"✅ File is really opened: {file\_path} by {len(processes)} processes")

else:

if file\_path in self.verified\_open\_files:

self.verified\_open\_files.remove(file\_path)

self.logger.debug(f"📁 File no longer opened: {file\_path}")

return is\_opened

except Exception as e:

self.logger.debug(f"Error checking if file is opened: {e}")

return True

def \_get\_processes\_using\_file(self, file\_path: str) -> list:

"""Возвращает список процессов, использующих файл"""

if not psutil:

return []

processes = []

try:

# Нормализуем путь для сравнения

normalized\_path = os.path.normpath(file\_path).lower()

for proc in psutil.process\_iter(['pid', 'name', 'username', 'open\_files']):

try:

open\_files = proc.info.get('open\_files')

if open\_files is None:

continue

for file in open\_files:

# Нормализуем путь открытого файла для сравнения

open\_file\_path = os.path.normpath(file.path).lower()

if open\_file\_path == normalized\_path:

# НОРМАЛИЗУЕМ имя пользователя процесса

process\_username = self.\_normalize\_username(proc.info.get('username', 'unknown'))

processes.append({

'pid': proc.pid,

'name': proc.info['name'],

'username': process\_username

})

break

except (psutil.NoSuchProcess, psutil.AccessDenied, FileNotFoundError):

continue

except Exception as e:

self.logger.debug(f"Error getting processes for {file\_path}: {e}")

return processes

def \_get\_file\_modifier\_safe(self, file\_path: str, event\_type: str) -> str:

"""Безопасное получение модификатора файла"""

try:

if not os.path.exists(file\_path):

return getpass.getuser()

return self.\_get\_file\_modifier(file\_path)

except Exception as e:

self.logger.warning(f"Failed to get file modifier for {file\_path}, using current user: {e}")

return getpass.getuser()

def \_get\_file\_modifier(self, file\_path: str) -> str:

"""Получает пользователя, изменившего файла"""

try:

if platform.system() == 'Windows' and win32security is not None:

sd = win32security.GetFileSecurity(file\_path, win32security.OWNER\_SECURITY\_INFORMATION)

owner\_sid = sd.GetSecurityDescriptorOwner()

name, domain, \_ = win32security.LookupAccountSid(None, owner\_sid)

return f"{domain}\\{name}"

else:

return getpass.getuser()

except Exception as e:

self.logger.error(f"Failed to get file modifier for {file\_path}: {e}")

return getpass.getuser()

def \_handle\_file\_created(self, file\_path: str, username: str, file\_hash: str) -> bool:

"""Обрабатывает создание файла"""

if file\_path in self.file\_renames.values() or file\_path in self.file\_move\_chains.values():

self.logger.debug(f"Ignoring created event for moved file: {file\_path}")

return True

self.logger.info(f"📄 Main file created: {file\_path} by {username}")

session\_data = self.session\_manager.smart\_create\_session(file\_path, username, file\_hash)

if session\_data.get('resume\_count', 0) > 0:

self.stats['sessions\_resumed'] += 1

self.logger.info(f"Session resumed for {file\_path} (resume count: {session\_data['resume\_count']})")

else:

self.stats['sessions\_created'] += 1

event\_data = {

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'event\_type': 'created',

'file\_hash': file\_hash,

'user\_id': username,

'session\_id': session\_data['session\_id'],

'resume\_count': session\_data.get('resume\_count', 0),

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"Failed to send created event for {file\_path}: {event\_data}")

return success

def \_handle\_file\_modified(self, file\_path: str, username: str, file\_hash: str) -> bool:

"""Обрабатывает изменение файла"""

if not self.\_is\_file\_really\_opened(file\_path):

self.logger.debug(f"📁 Main file not actually opened, skipping modified event: {file\_path}")

return True

self.logger.debug(f"📝 Main file modified: {file\_path} by {username}")

session\_data = self.session\_manager.smart\_create\_session(file\_path, username, file\_hash)

event\_data = {

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'event\_type': 'modified',

'file\_hash': file\_hash,

'user\_id': username,

'session\_id': session\_data['session\_id'],

'resume\_count': session\_data.get('resume\_count', 0),

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"Failed to send modified event for {file\_path}: {event\_data}")

return success

def \_handle\_file\_deleted(self, file\_path: str, username: str) -> bool:

"""Обрабатывает удаление файла"""

if file\_path in self.file\_renames or file\_path in self.file\_move\_chains:

self.logger.debug(f"📦 File moved, closing session for: {file\_path}")

session\_data = self.session\_manager.close\_session(file\_path, username)

if session\_data:

self.logger.info(f"✅ Closed session for moved file: {file\_path}")

if file\_path in self.file\_renames:

del self.file\_renames[file\_path]

if file\_path in self.file\_move\_chains:

del self.file\_move\_chains[file\_path]

return True

self.logger.info(f"🗑️ Main file deleted: {file\_path} by {username}")

self.stats['files\_deleted'] += 1

if file\_path in self.open\_files:

del self.open\_files[file\_path]

if file\_path in self.file\_renames:

del self.file\_renames[file\_path]

if file\_path in self.file\_move\_chains:

del self.file\_move\_chains[file\_path]

if file\_path in self.verified\_open\_files:

self.verified\_open\_files.remove(file\_path)

# Очищаем из temp\_to\_main\_map

keys\_to\_remove = []

for temp\_path, main\_path in self.temp\_to\_main\_map.items():

if temp\_path == file\_path or main\_path == file\_path:

keys\_to\_remove.append(temp\_path)

for key in keys\_to\_remove:

del self.temp\_to\_main\_map[key]

# Очищаем из main\_file\_tracking

if file\_path in self.main\_file\_tracking:

del self.main\_file\_tracking[file\_path]

session\_data = self.session\_manager.close\_session(file\_path, username)

if not session\_data:

self.logger.info(f"🔄 Using forced session close for: {file\_path}")

closed\_sessions = self.session\_manager.close\_all\_sessions\_for\_file(file\_path)

if closed\_sessions:

session\_data = closed\_sessions[0]

self.logger.info(f"✅ Forced close: closed {len(closed\_sessions)} sessions")

if session\_data:

self.logger.info(f"✅ Successfully closed session for deleted file: {file\_path}")

event\_data = {

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'event\_type': 'deleted',

'user\_id': username,

'session\_id': session\_data['session\_id'],

'resume\_count': session\_data.get('resume\_count', 0),

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

if not success:

self.logger.error(f"❌ Failed to send deleted event for: {file\_path}")

return success

else:

self.logger.warning(f"⚠️ No session found for deleted file: {file\_path}")

event\_data = {

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'event\_type': 'deleted',

'user\_id': username,

'event\_timestamp': datetime.now().isoformat()

}

success = self.api\_client.send\_event(event\_data)

return success

def \_update\_open\_file\_tracking(self, file\_path: str, username: str, event\_type: str):

"""Обновляет информацию об открытых файлах"""

if not psutil:

return

try:

current\_processes = self.\_get\_processes\_using\_file(file\_path)

current\_time = datetime.now()

if current\_processes:

# Файл открыт - обновляем информацию

self.open\_files[file\_path] = {

'username': username,

'processes': current\_processes,

'last\_activity': current\_time,

'last\_checked': current\_time,

'event\_type': event\_type

}

self.logger.debug(f"File {file\_path} is open in {len(current\_processes)} processes")

else:

# Файл больше не открыт - проверяем нужно ли закрыть сессию

if file\_path in self.open\_files:

file\_info = self.open\_files[file\_path]

time\_since\_last\_activity = current\_time - file\_info['last\_activity']

# Закрываем сессию только если прошло достаточно времени с последней активности

if time\_since\_last\_activity > timedelta(seconds=5):

self.logger.info(f"File {file\_path} is no longer open, closing session")

# Вычисляем финальный хеш если файл существует

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Закрываем сессию

self.\_handle\_file\_closed(file\_path, file\_info['username'], file\_hash)

del self.open\_files[file\_path]

self.stats['files\_closed'] += 1

else:

# Обновляем время проверки, но не закрываем сессию

self.open\_files[file\_path]['last\_checked'] = current\_time

except Exception as e:

self.logger.error(f"Error updating open file tracking for {file\_path}: {e}")

def \_handle\_file\_closed(self, file\_path: str, username: str, file\_hash: str) -> bool:

"""Обрабатывает закрытие файла"""

self.logger.info(f"File closed: {file\_path} by {username}")

# Закрываем сессию в SessionManager

session\_data = self.session\_manager.close\_session(file\_path, username, file\_hash)

if session\_data:

# ВАЖНО: проверяем что ended\_at установлен

if 'ended\_at' not in session\_data or session\_data['ended\_at'] is None:

self.logger.error(f"❌ Session closed but ended\_at is not set for {file\_path}")

# Устанавливаем ended\_at если его нет

session\_data['ended\_at'] = datetime.now()

self.logger.info(f"✅ Manually set ended\_at to: {session\_data['ended\_at']}")

session\_duration = (session\_data['ended\_at'] - session\_data['started\_at']).total\_seconds()

event\_data = {

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'event\_type': 'closed',

'file\_hash': file\_hash,

'user\_id': username,

'session\_id': session\_data['session\_id'],

'resume\_count': session\_data.get('resume\_count', 0),

'session\_duration': session\_duration,

'event\_timestamp': session\_data['ended\_at'].isoformat()

}

success = self.api\_client.send\_event(event\_data)

if success:

self.logger.info(f"✅ Successfully closed session for {file\_path} (duration: {session\_duration:.1f}s, ended\_at: {session\_data['ended\_at']})")

else:

self.logger.error(f"❌ Failed to send closed event for {file\_path}")

return success

else:

self.logger.warning(f"No active session found for closed file: {file\_path}")

return True

def check\_open\_files(self):

"""Периодически проверяет состояние открытых файлов"""

if not psutil:

return

try:

current\_time = datetime.now()

check\_interval = timedelta(seconds=30)

if current\_time - self.last\_open\_files\_check < check\_interval:

return

self.last\_open\_files\_check = current\_time

files\_to\_close = []

for file\_path, file\_info in list(self.open\_files.items()):

# Проверяем, открыт ли файл все еще

current\_processes = self.\_get\_processes\_using\_file(file\_path)

if not current\_processes:

# Файл больше не открыт - проверяем время с последней активности

time\_since\_last\_activity = current\_time - file\_info['last\_activity']

# Закрываем сессию только если прошло достаточно времени

if time\_since\_last\_activity > timedelta(seconds=5):

files\_to\_close.append((file\_path, file\_info))

else:

# Обновляем время проверки

file\_info['last\_checked'] = current\_time

else:

# Файл все еще открыт - обновляем информацию

file\_info['processes'] = current\_processes

file\_info['last\_checked'] = current\_time

# Закрываем сессии для файлов, которые больше не открыты

for file\_path, file\_info in files\_to\_close:

self.logger.info(f"Detected file closure: {file\_path}")

# Вычисляем финальный хеш если файл существует

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Закрываем сессию

self.\_handle\_file\_closed(file\_path, file\_info['username'], file\_hash)

del self.open\_files[file\_path]

self.stats['files\_closed'] += 1

except Exception as e:

self.logger.error(f"Error checking open files: {e}")

def check\_expired\_sessions(self):

"""Проверяет и закрывает просроченные сессии"""

try:

self.logger.debug("🔍 Starting expired sessions check...")

expired\_sessions = self.session\_manager.check\_and\_close\_expired\_sessions()

closed\_count = 0

for session\_data in expired\_sessions:

file\_path = session\_data['file\_path']

username = session\_data['username']

self.stats['expired\_sessions'] += 1

closed\_count += 1

# ПРОВЕРЯЕМ ЧТО ended\_at УСТАНОВЛЕНО

if 'ended\_at' not in session\_data or session\_data['ended\_at'] is None:

self.logger.error(f"❌ Session closed but ended\_at is None for: {file\_path}")

continue

# Вычисляем финальный хеш если файл существует

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

# Отправляем событие closed для expired сессии

event\_data = {

'file\_path': file\_path,

'file\_name': session\_data.get('file\_name', os.path.basename(file\_path)),

'event\_type': 'closed',

'file\_hash': file\_hash,

'user\_id': username,

'session\_id': session\_data['session\_id'],

'resume\_count': session\_data.get('resume\_count', 0),

'session\_duration': (session\_data['ended\_at'] - session\_data['started\_at']).total\_seconds(),

'event\_timestamp': session\_data['ended\_at'].isoformat()

}

success = self.api\_client.send\_event(event\_data)

if success:

self.logger.info(f"✅ Closed expired session: {file\_path} (ended\_at: {session\_data['ended\_at']})")

else:

self.logger.error(f"❌ Failed to send closed event for: {file\_path}")

return closed\_count

except Exception as e:

self.logger.error(f"❌ Error checking expired sessions: {e}")

return 0

def cleanup\_orphaned\_sessions(self):

"""Очищает сессии для файлов, которые больше не существуют"""

expired\_sessions = []

for session\_key, session\_data in list(self.session\_manager.active\_sessions.items()):

file\_path = session\_data['file\_path']

username = session\_data['username']

if not os.path.exists(file\_path):

self.logger.info(f"Closing orphaned session for deleted file: {file\_path}")

closed\_session = self.session\_manager.close\_session(file\_path, username)

if closed\_session:

expired\_sessions.append(closed\_session)

event\_data = {

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'event\_type': 'deleted',

'user\_id': username,

'session\_id': closed\_session['session\_id'],

'resume\_count': closed\_session.get('resume\_count', 0),

'event\_timestamp': datetime.now().isoformat()

}

self.api\_client.send\_event(event\_data)

return expired\_sessions

def get\_stats(self) -> Dict[str, Any]:

"""Возвращает статистику обработки"""

session\_stats = self.session\_manager.get\_session\_stats()

return {

\*\*self.stats,

\*\*session\_stats,

'open\_files\_tracking': len(self.open\_files),

'file\_move\_chains': len(self.file\_move\_chains),

'verified\_open\_files': len(self.verified\_open\_files),

'temp\_to\_main\_mappings': len(self.temp\_to\_main\_map),

'main\_files\_tracked': len(self.main\_file\_tracking)

}

def cleanup(self):

"""Очищает ресурсы"""

# Очищаем expired сессии

expired\_sessions = self.session\_manager.cleanup\_expired\_sessions(self)

for session\_data in expired\_sessions:

file\_path = session\_data['file\_path']

username = session\_data['username']

file\_hash = None

if os.path.exists(file\_path) and self.config.get('hashing', {}).get('enabled', True):

file\_hash = self.hash\_calculator.calculate\_file\_hash\_with\_retry(file\_path)

self.\_handle\_file\_closed(file\_path, username, file\_hash)

# Проверяем открытые файлы

self.check\_open\_files()

# Очищаем orphaned сессии

self.cleanup\_orphaned\_sessions(), timedelta

from typing import Dict, Optional, List

from shared.logger import setup\_logger

class SessionManager:

def \_\_iniimport os

import uuid

from datetime import datetime, timedelta

from typing import Dict, Optional, List

from shared.logger import setup\_logger

class SessionManager:

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

self.active\_sessions: Dict[str, Dict] = {} # file\_path:username -> session\_data

self.closed\_sessions: Dict[str, List[Dict]] = {} # История закрытых сессий

self.logger = setup\_logger(\_\_name\_\_)

self.config = {}

def set\_config(self, config: dict):

"""Устанавливает конфигурацию"""

self.config = config

timeout = self.config.get('session\_timeout\_minutes', 1)

self.logger.info(f"⚙️ Session config: timeout={timeout}min, max\_age={self.config.get('max\_session\_hours', 3)}h")

def \_get\_session\_key(self, file\_path: str, username: str) -> str:

"""Генерирует ключ сессии"""

return f"{file\_path}:{username}"

def \_find\_recently\_closed(self, session\_key: str, hours: int = 1) -> Optional[Dict]:

"""Находит недавно закрытую сессию для возможного возобновления"""

if session\_key not in self.closed\_sessions:

return None

closed\_sessions = self.closed\_sessions[session\_key]

if not closed\_sessions:

return None

# Берем последнюю закрытую сессию

last\_session = closed\_sessions[-1]

# Проверяем, закрыта ли она в пределах указанного времени

if 'ended\_at' in last\_session and last\_session['ended\_at']:

time\_since\_close = datetime.now() - last\_session['ended\_at']

if time\_since\_close <= timedelta(hours=hours):

return last\_session

return None

def \_resume\_session(self, session\_data: Dict, file\_hash: str = None) -> Dict:

"""Возобновляет существующую сессию"""

session\_key = self.\_get\_session\_key(session\_data['file\_path'], session\_data['username'])

# Обновляем данные сессии

resumed\_session = session\_data.copy()

resumed\_session['last\_activity'] = datetime.now()

resumed\_session['resumed\_at'] = datetime.now()

resumed\_session['resume\_count'] = resumed\_session.get('resume\_count', 0) + 1

resumed\_session['hash\_before'] = file\_hash

# Убираем поля окончания, т.к. сессия снова активна

resumed\_session.pop('ended\_at', None)

resumed\_session.pop('hash\_after', None)

# Возвращаем в активные сессии

self.active\_sessions[session\_key] = resumed\_session

# Удаляем из истории закрытых, если она там есть

if session\_key in self.closed\_sessions and session\_data in self.closed\_sessions[session\_key]:

self.closed\_sessions[session\_key].remove(session\_data)

self.logger.info(f"🔄 Resumed session for {resumed\_session['file\_path']}")

return resumed\_session

def get\_active\_session(self, file\_path: str, username: str) -> Optional[Dict]:

"""Возвращает активную сессию для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.get(session\_key)

if session\_data:

# Проверяем не истекла ли сессия

if self.\_is\_session\_expired(session\_data):

# Сессия истекла - закрываем ее

self.logger.info(f"🕒 Session expired, closing: {file\_path}")

self.close\_session(file\_path, username)

return None

# Обновляем время последней активности

session\_data['last\_activity'] = datetime.now()

return session\_data

def \_is\_session\_expired(self, session\_data: Dict) -> bool:

"""Проверяет истекла ли сессия"""

timeout\_minutes = self.config.get('session\_timeout\_minutes', 1)

max\_age\_hours = self.config.get('max\_session\_hours', 3)

last\_activity = session\_data['last\_activity']

session\_age = datetime.now() - session\_data['started\_at']

time\_since\_activity = datetime.now() - last\_activity

timeout\_seconds = timeout\_minutes \* 60

# Проверяем таймаут активности

if time\_since\_activity.total\_seconds() > timeout\_seconds:

self.logger.info(f"🕒 Session expired by timeout: {session\_data['file\_path']}, inactive for {time\_since\_activity.total\_seconds():.1f}s > {timeout\_seconds}s")

return True

# Проверяем максимальный возраст сессии

if session\_age.total\_seconds() > (max\_age\_hours \* 3600):

self.logger.info(f"📅 Session expired by max age: {session\_data['file\_path']}, age: {session\_age.total\_seconds()/3600:.1f}h")

return True

return False

def check\_and\_close\_expired\_sessions(self) -> List[Dict]:

"""Проверяет и закрывает все просроченные сессии - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

expired\_sessions = []

total\_sessions = len(self.active\_sessions)

if total\_sessions == 0:

return expired\_sessions

self.logger.info(f"🔍 Checking {total\_sessions} active sessions for expiration...")

# Создаем копию списка для безопасной итерации

sessions\_to\_check = list(self.active\_sessions.items())

for session\_key, session\_data in sessions\_to\_check:

try:

if self.\_is\_session\_expired(session\_data):

file\_path = session\_data['file\_path']

username = session\_data['username']

time\_since\_activity = datetime.now() - session\_data['last\_activity']

self.logger.info(f"🔒 Closing expired session: {file\_path} (inactive: {time\_since\_activity.total\_seconds():.1f}s)")

# Закрываем сессию и получаем данные с ended\_at

closed\_data = self.close\_session(file\_path, username)

if closed\_data:

expired\_sessions.append(closed\_data)

self.logger.info(f"✅ Session closed with ended\_at: {closed\_data['ended\_at']}")

except Exception as e:

self.logger.error(f"❌ Error checking session {session\_key}: {e}")

if expired\_sessions:

self.logger.info(f"✅ Closed {len(expired\_sessions)} expired sessions")

else:

self.logger.debug(f"📊 All {total\_sessions} sessions are active")

return expired\_sessions

def smart\_create\_session(self, file\_path: str, username: str, file\_hash: str = None, resume\_window\_hours: int = 1) -> Dict:

"""Умное создание сессии с возможностью возобновления недавно закрытой сессии"""

# Сначала проверяем активную сессию

active\_session = self.get\_active\_session(file\_path, username)

if active\_session:

return active\_session

# Пытаемся найти недавно закрытую сессию для возобновления

session\_key = self.\_get\_session\_key(file\_path, username)

recently\_closed = self.\_find\_recently\_closed(session\_key, resume\_window\_hours)

if recently\_closed:

return self.\_resume\_session(recently\_closed, file\_hash)

else:

return self.create\_session(file\_path, username, file\_hash)

def create\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Создает новую сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = {

'session\_id': str(uuid.uuid4()),

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'username': username,

'started\_at': datetime.now(),

'last\_activity': datetime.now(),

'hash\_before': file\_hash,

'events': [],

'resume\_count': 0

}

self.active\_sessions[session\_key] = session\_data

self.logger.info(f"✅ Created session for {file\_path}")

return session\_data

def update\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Обновляет существующую сессию или создает новую"""

session\_data = self.get\_active\_session(file\_path, username)

if session\_data:

# Обновляем существующую сессию

if file\_hash:

session\_data['hash\_after'] = file\_hash

session\_data['last\_activity'] = datetime.now()

self.logger.debug(f"📝 Updated session for {file\_path}")

else:

# Создаем новую сессию

session\_data = self.create\_session(file\_path, username, file\_hash)

return session\_data

def close\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Закрывает сессию - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.pop(session\_key, None)

if session\_data:

# Устанавливаем время окончания - ВАЖНО: ДОБАВЛЕНО ВРЕМЯ ЗАКРЫТИЯ

ended\_at = datetime.now()

session\_data['ended\_at'] = ended\_at

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем в историю закрытых сессий

if session\_key not in self.closed\_sessions:

self.closed\_sessions[session\_key] = []

self.closed\_sessions[session\_key].append(session\_data)

# Ограничиваем историю

if len(self.closed\_sessions[session\_key]) > 10:

self.closed\_sessions[session\_key] = self.closed\_sessions[session\_key][-10:]

# Логируем с информацией о времени сессии

session\_duration = ended\_at - session\_data['started\_at']

self.logger.info(f"🔒 Closed session for {file\_path} (duration: {session\_duration.total\_seconds():.1f}s, ended\_at: {ended\_at})")

return session\_data

else:

self.logger.debug(f"❌ No active session found for: {file\_path} (user: {username})")

return None

def close\_all\_sessions\_for\_file(self, file\_path: str) -> List[Dict]:

"""Принудительно закрывает все сессии для указанного файла"""

closed\_sessions = []

sessions\_to\_close = []

for session\_key, session\_data in list(self.active\_sessions.items()):

if session\_data['file\_path'] == file\_path:

sessions\_to\_close.append((session\_data['file\_path'], session\_data['username']))

self.logger.info(f"🔍 Found {len(sessions\_to\_close)} sessions to close for: {file\_path}")

for file\_path, username in sessions\_to\_close:

session\_data = self.close\_session(file\_path, username)

if session\_data:

closed\_sessions.append(session\_data)

return closed\_sessions

def transfer\_session(self, old\_file\_path: str, new\_file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Переносит сессию со старого пути на новый - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

session\_key\_old = self.\_get\_session\_key(old\_file\_path, username)

session\_data = self.active\_sessions.pop(session\_key\_old, None)

if session\_data:

# Обновляем путь файла в данных сессии

session\_data['file\_path'] = new\_file\_path

session\_data['file\_name'] = os.path.basename(new\_file\_path)

session\_data['last\_activity'] = datetime.now()

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем с новым ключом

session\_key\_new = self.\_get\_session\_key(new\_file\_path, username)

self.active\_sessions[session\_key\_new] = session\_data

# Также обновляем в истории закрытых сессий, если нужно

if session\_key\_old in self.closed\_sessions:

# Переносим историю на новый ключ

self.closed\_sessions[session\_key\_new] = self.closed\_sessions.pop(session\_key\_old)

self.logger.info(f"🔄 Transferred session from {old\_file\_path} to {new\_file\_path}")

return session\_data

return None

def cleanup\_expired\_sessions(self, event\_handler) -> list:

"""Очищает просроченные сессии"""

return self.check\_and\_close\_expired\_sessions()

def get\_session\_stats(self) -> Dict:

"""Возвращает статистику по сессиям"""

total\_resumes = sum(session.get('resume\_count', 0) for session in self.active\_sessions.values())

return {

'active\_sessions': len(self.active\_sessions),

'session\_keys': list(self.active\_sessions.keys()),

'total\_resumes': total\_resumes,

'closed\_sessions\_count': sum(len(sessions) for sessions in self.closed\_sessions.values())

}

def get\_session\_history(self, file\_path: str, username: str) -> List[Dict]:

"""Возвращает историю сессий для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

return self.closed\_sessions.get(session\_key, [])t\_\_(self):

self.active\_sessions: Dict[str, Dict] = {} # file\_path:username -> session\_data

self.closed\_sessions: Dict[str, List[Dict]] = {} # История закрытых сессий

self.logger = setup\_logger(\_\_name\_\_)

self.config = {}

def set\_config(self, config: dict):

"""Устанавливает конфигурацию"""

self.config = config

timeout = self.config.get('session\_timeout\_minutes', 1)

self.logger.info(f"⚙️ Session config: timeout={timeout}min, max\_age={self.config.get('max\_session\_hours', 3)}h")

def \_get\_session\_key(self, file\_path: str, username: str) -> str:

"""Генерирует ключ сессии"""

return f"{file\_path}:{username}"

def \_find\_recently\_closed(self, session\_key: str, hours: int = 1) -> Optional[Dict]:

"""Находит недавно закрытую сессию для возможного возобновления"""

if session\_key not in self.closed\_sessions:

return None

closed\_sessions = self.closed\_sessions[session\_key]

if not closed\_sessions:

return None

# Берем последнюю закрытую сессию

last\_session = closed\_sessions[-1]

# Проверяем, закрыта ли она в пределах указанного времени

if 'ended\_at' in last\_session and last\_session['ended\_at']:

time\_since\_close = datetime.now() - last\_session['ended\_at']

if time\_since\_close <= timedelta(hours=hours):

return last\_session

return None

def \_resume\_session(self, session\_data: Dict, file\_hash: str = None) -> Dict:

"""Возобновляет существующую сессию"""

session\_key = self.\_get\_session\_key(session\_data['file\_path'], session\_data['username'])

# Обновляем данные сессии

resumed\_session = session\_data.copy()

resumed\_session['last\_activity'] = datetime.now()

resumed\_session['resumed\_at'] = datetime.now()

resumed\_session['resume\_count'] = resumed\_session.get('resume\_count', 0) + 1

resumed\_session['hash\_before'] = file\_hash

# Убираем поля окончания, т.к. сессия снова активна

resumed\_session.pop('ended\_at', None)

resumed\_session.pop('hash\_after', None)

# Возвращаем в активные сессии

self.active\_sessions[session\_key] = resumed\_session

# Удаляем из истории закрытых, если она там есть

if session\_key in self.closed\_sessions and session\_data in self.closed\_sessions[session\_key]:

self.closed\_sessions[session\_key].remove(session\_data)

self.logger.info(f"🔄 Resumed session for {resumed\_session['file\_path']}")

return resumed\_session

def get\_active\_session(self, file\_path: str, username: str) -> Optional[Dict]:

"""Возвращает активную сессию для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.get(session\_key)

if session\_data:

# Проверяем не истекла ли сессия

if self.\_is\_session\_expired(session\_data):

# Сессия истекла - закрываем ее

self.logger.info(f"🕒 Session expired, closing: {file\_path}")

self.close\_session(file\_path, username)

return None

# Обновляем время последней активности

session\_data['last\_activity'] = datetime.now()

return session\_data

def \_is\_session\_expired(self, session\_data: Dict) -> bool:

"""Проверяет истекла ли сессия"""

timeout\_minutes = self.config.get('session\_timeout\_minutes', 1)

max\_age\_hours = self.config.get('max\_session\_hours', 3)

last\_activity = session\_data['last\_activity']

session\_age = datetime.now() - session\_data['started\_at']

time\_since\_activity = datetime.now() - last\_activity

timeout\_seconds = timeout\_minutes \* 60

# Проверяем таймаут активности

if time\_since\_activity.total\_seconds() > timeout\_seconds:

self.logger.info(f"🕒 Session expired by timeout: {session\_data['file\_path']}, inactive for {time\_since\_activity.total\_seconds():.1f}s > {timeout\_seconds}s")

return True

# Проверяем максимальный возраст сессии

if session\_age.total\_seconds() > (max\_age\_hours \* 3600):

self.logger.info(f"📅 Session expired by max age: {session\_data['file\_path']}, age: {session\_age.total\_seconds()/3600:.1f}h")

return True

return False

def check\_and\_close\_expired\_sessions(self) -> List[Dict]:

"""Проверяет и закрывает все просроченные сессии - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

expired\_sessions = []

total\_sessions = len(self.active\_sessions)

if total\_sessions == 0:

return expired\_sessions

self.logger.info(f"🔍 Checking {total\_sessions} active sessions for expiration...")

# Создаем копию списка для безопасной итерации

sessions\_to\_check = list(self.active\_sessions.items())

for session\_key, session\_data in sessions\_to\_check:

try:

if self.\_is\_session\_expired(session\_data):

file\_path = session\_data['file\_path']

username = session\_data['username']

time\_since\_activity = datetime.now() - session\_data['last\_activity']

self.logger.info(f"🔒 Closing expired session: {file\_path} (inactive: {time\_since\_activity.total\_seconds():.1f}s)")

# Закрываем сессию и получаем данные с ended\_at

closed\_data = self.close\_session(file\_path, username)

if closed\_data:

expired\_sessions.append(closed\_data)

self.logger.info(f"✅ Session closed with ended\_at: {closed\_data['ended\_at']}")

except Exception as e:

self.logger.error(f"❌ Error checking session {session\_key}: {e}")

if expired\_sessions:

self.logger.info(f"✅ Closed {len(expired\_sessions)} expired sessions")

else:

self.logger.debug(f"📊 All {total\_sessions} sessions are active")

return expired\_sessions

def smart\_create\_session(self, file\_path: str, username: str, file\_hash: str = None, resume\_window\_hours: int = 1) -> Dict:

"""Умное создание сессии с возможностью возобновления недавно закрытой сессии"""

# Сначала проверяем активную сессию

active\_session = self.get\_active\_session(file\_path, username)

if active\_session:

return active\_session

# Пытаемся найти недавно закрытую сессию для возобновления

session\_key = self.\_get\_session\_key(file\_path, username)

recently\_closed = self.\_find\_recently\_closed(session\_key, resume\_window\_hours)

if recently\_closed:

return self.\_resume\_session(recently\_closed, file\_hash)

else:

return self.create\_session(file\_path, username, file\_hash)

def create\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Создает новую сессию"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = {

'session\_id': str(uuid.uuid4()),

'file\_path': file\_path,

'file\_name': os.path.basename(file\_path),

'username': username,

'started\_at': datetime.now(),

'last\_activity': datetime.now(),

'hash\_before': file\_hash,

'events': [],

'resume\_count': 0

}

self.active\_sessions[session\_key] = session\_data

self.logger.info(f"✅ Created session for {file\_path}")

return session\_data

def update\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Dict:

"""Обновляет существующую сессию или создает новую"""

session\_data = self.get\_active\_session(file\_path, username)

if session\_data:

# Обновляем существующую сессию

if file\_hash:

session\_data['hash\_after'] = file\_hash

session\_data['last\_activity'] = datetime.now()

self.logger.debug(f"📝 Updated session for {file\_path}")

else:

# Создаем новую сессию

session\_data = self.create\_session(file\_path, username, file\_hash)

return session\_data

def close\_session(self, file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Закрывает сессию - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

session\_key = self.\_get\_session\_key(file\_path, username)

session\_data = self.active\_sessions.pop(session\_key, None)

if session\_data:

# Устанавливаем время окончания - ВАЖНО: ДОБАВЛЕНО ВРЕМЯ ЗАКРЫТИЯ

ended\_at = datetime.now()

session\_data['ended\_at'] = ended\_at

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем в историю закрытых сессий

if session\_key not in self.closed\_sessions:

self.closed\_sessions[session\_key] = []

self.closed\_sessions[session\_key].append(session\_data)

# Ограничиваем историю

if len(self.closed\_sessions[session\_key]) > 10:

self.closed\_sessions[session\_key] = self.closed\_sessions[session\_key][-10:]

# Логируем с информацией о времени сессии

session\_duration = ended\_at - session\_data['started\_at']

self.logger.info(f"🔒 Closed session for {file\_path} (duration: {session\_duration.total\_seconds():.1f}s, ended\_at: {ended\_at})")

return session\_data

else:

self.logger.debug(f"❌ No active session found for: {file\_path} (user: {username})")

return None

def close\_all\_sessions\_for\_file(self, file\_path: str) -> List[Dict]:

"""Принудительно закрывает все сессии для указанного файла"""

closed\_sessions = []

sessions\_to\_close = []

for session\_key, session\_data in list(self.active\_sessions.items()):

if session\_data['file\_path'] == file\_path:

sessions\_to\_close.append((session\_data['file\_path'], session\_data['username']))

self.logger.info(f"🔍 Found {len(sessions\_to\_close)} sessions to close for: {file\_path}")

for file\_path, username in sessions\_to\_close:

session\_data = self.close\_session(file\_path, username)

if session\_data:

closed\_sessions.append(session\_data)

return closed\_sessions

def transfer\_session(self, old\_file\_path: str, new\_file\_path: str, username: str, file\_hash: str = None) -> Optional[Dict]:

"""Переносит сессию со старого пути на новый - ИСПРАВЛЕННАЯ ВЕРСИЯ"""

session\_key\_old = self.\_get\_session\_key(old\_file\_path, username)

session\_data = self.active\_sessions.pop(session\_key\_old, None)

if session\_data:

# Обновляем путь файла в данных сессии

session\_data['file\_path'] = new\_file\_path

session\_data['file\_name'] = os.path.basename(new\_file\_path)

session\_data['last\_activity'] = datetime.now()

if file\_hash:

session\_data['hash\_after'] = file\_hash

# Сохраняем с новым ключом

session\_key\_new = self.\_get\_session\_key(new\_file\_path, username)

self.active\_sessions[session\_key\_new] = session\_data

# Также обновляем в истории закрытых сессий, если нужно

if session\_key\_old in self.closed\_sessions:

# Переносим историю на новый ключ

self.closed\_sessions[session\_key\_new] = self.closed\_sessions.pop(session\_key\_old)

self.logger.info(f"🔄 Transferred session from {old\_file\_path} to {new\_file\_path}")

return session\_data

return None

def cleanup\_expired\_sessions(self, event\_handler) -> list:

"""Очищает просроченные сессии"""

return self.check\_and\_close\_expired\_sessions()

def get\_session\_stats(self) -> Dict:

"""Возвращает статистику по сессиям"""

total\_resumes = sum(session.get('resume\_count', 0) for session in self.active\_sessions.values())

return {

'active\_sessions': len(self.active\_sessions),

'session\_keys': list(self.active\_sessions.keys()),

'total\_resumes': total\_resumes,

'closed\_sessions\_count': sum(len(sessions) for sessions in self.closed\_sessions.values())

}

def get\_session\_history(self, file\_path: str, username: str) -> List[Dict]:

"""Возвращает историю сессий для файла и пользователя"""

session\_key = self.\_get\_session\_key(file\_path, username)

return self.closed\_sessions.get(session\_key, [])

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

91027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False

import os

import fnmatch

import re

from typing import List

from shared.logger import setup\_loggerhttps://www.citilink.ru/product/planshet-xiaomi-redmi-pad-2-g100-8c-8gb-256gb-11-ips-1340x800-4g-zelen-2112203/?referrer=reattribution%3D1&utm\_term=---autotargeting&utm\_medium=cpc&utm\_campaign=cities-srch-cat-perfomans-epk\_all&utm\_source=yandex&utm\_content=%7Cc%3A105534431%7Cg%3A5388562707%7Cb%3A1832012770338417349%7Ck%3A49891027085%7Cst%3Asearch%7Ca%3Ano%7Cs%3Ayandex.ru%7Ct%3Aother%7Cp%3A1%7Cr%3A49891027085%7Cdev%3Adesktop|cgci:0&ybaip=1&yclid=7699611067276328959

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return Falseimport os

import fnmatch

import re

from typing import List

from shared.logger import setup\_logger

class FileValidator:

def \_\_init\_\_(self, config: dict):

self.config = config

self.logger = setup\_logger(\_\_name\_\_)

# Компилируем паттерны для лучшей производительности

self.ignore\_patterns = self.config.get('ignore\_patterns', [])

self.ignore\_extensions = self.config.get('ignore\_extensions', [])

self.ignore\_dirs = self.config.get('ignore\_dirs', [])

# ДОБАВЛЕНО: Расширенный список временных паттернов для Excel

self.excel\_temp\_patterns = [

'~$\*', '~wr\*', '~rf\*', '\*.tmp', '\*.temp',

'~\*', '.\*~', '\*.~\*', '\*~\*.tmp'

]

def should\_monitor\_file(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл"""

if not os.path.isfile(file\_path):

return False

filename = os.path.basename(file\_path)

file\_ext = os.path.splitext(filename)[1].lower()

# Проверяем игнорируемые паттерны (с улучшенной логикой)

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring Excel temporary file: {filename}")

return False

return True

def should\_monitor\_file\_by\_name(self, file\_path: str) -> bool:

"""Определяет нужно ли отслеживать файл только по имени (когда файл уже удален)"""

filename = os.path.basename(file\_path)

# Проверяем игнорируемые паттерны

if self.\_matches\_ignore\_patterns(filename, file\_path):

self.logger.debug(f"Ignoring deleted file due to pattern: {filename}")

return False

# Проверяем игнорируемые расширения

file\_ext = os.path.splitext(filename)[1].lower()

if file\_ext in self.ignore\_extensions:

self.logger.debug(f"Ignoring deleted file due to extension: {filename}")

return False

# Проверяем разрешенные расширения

allowed\_extensions = self.config.get('file\_extensions', [])

if allowed\_extensions and file\_ext not in allowed\_extensions:

self.logger.debug(f"Ignoring deleted file - extension not in allowed list: {filename}")

return False

# Проверяем игнорируемые директории в пути

if self.\_contains\_ignore\_dirs(file\_path):

self.logger.debug(f"Ignoring deleted file in excluded directory: {file\_path}")

return False

# ДОПОЛНИТЕЛЬНАЯ ПРОВЕРКА: Явно игнорируем временные файлы Excel

if self.\_is\_excel\_temporary\_file(filename):

self.logger.debug(f"Ignoring deleted Excel temporary file: {filename}")

return False

return True

def \_matches\_ignore\_patterns(self, filename: str, file\_path: str) -> bool:

"""Проверяет совпадение с игнорируемыми паттернами"""

for pattern in self.ignore\_patterns:

# Для паттернов, начинающихся с \* (например, "\*.tmp")

if pattern.startswith('\*') and filename.endswith(pattern[1:]):

return True

# Для паттернов, заканчивающихся на \* (например, "~\*")

elif pattern.endswith('\*') and filename.startswith(pattern[:-1]):

return True

# Для паттернов с \* в середине

elif '\*' in pattern:

if fnmatch.fnmatch(filename, pattern):

return True

# Точное совпадение

elif filename == pattern:

return True

# Проверка по регулярному выражению для сложных паттернов

elif self.\_is\_regex\_match(filename, pattern):

return True

return False

def \_is\_regex\_match(self, filename: str, pattern: str) -> bool:

"""Проверяет совпадение с использованием упрощенных regex-паттернов"""

try:

# Преобразуем fnmatch-паттерн в regex

regex\_pattern = fnmatch.translate(pattern)

return re.match(regex\_pattern, filename) is not None

except re.error:

return False

def \_contains\_ignore\_dirs(self, file\_path: str) -> bool:

"""Проверяет содержит ли путь игнорируемые директории"""

if not self.ignore\_dirs:

return False

# Нормализуем путь для кроссплатформенности

normalized\_path = os.path.normpath(file\_path).lower()

path\_parts = normalized\_path.split(os.sep)

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем, содержится ли игнорируемая директория в пути

if ignore\_dir\_normalized in path\_parts:

return True

return False

def \_is\_excel\_temporary\_file(self, filename: str) -> bool:

"""Определяет является ли файл временным файлом Excel"""

filename\_lower = filename.lower()

# Паттерны временных файлов Excel

excel\_temp\_indicators = [

'~$', # Автосохранение Excel

'~wr', # Временные файлы записи

'~rf', # Временные файлы восстановления

'.tmp', # Временные файлы

'.temp', # Временные файлы

'~', # Файлы с тильдой

]

# Проверяем все паттерны

for pattern in excel\_temp\_indicators:

if pattern in filename\_lower:

return True

# Проверяем сложные паттерны типа "filename~RFabc123.tmp"

if '~' in filename\_lower and (filename\_lower.endswith('.tmp') or filename\_lower.endswith('.temp')):

return True

return False

def get\_monitorable\_files(self, directory: str) -> List[str]:

"""Возвращает список файлов для мониторинга в директории"""

monitorable\_files = []

try:

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

# Исключаем игнорируемые директории из дальнейшего обхода

dirs[:] = [d for d in dirs if not self.\_should\_ignore\_dir(os.path.join(root, d))]

for file in files:

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

if self.should\_monitor\_file(file\_path):

monitorable\_files.append(file\_path)

except Exception as e:

self.logger.error(f"Error scanning directory {directory}: {e}")

return monitorable\_files

def \_should\_ignore\_dir(self, dir\_path: str) -> bool:

"""Определяет нужно ли игнорировать директорию"""

dir\_name = os.path.basename(dir\_path)

# Проверяем паттерны имен директорий

for pattern in self.ignore\_patterns:

if self.\_matches\_ignore\_patterns(dir\_name, dir\_path):

return True

# Проверяем конкретные директории из ignore\_dirs

if self.ignore\_dirs:

normalized\_path = os.path.normpath(dir\_path).lower()

for ignore\_dir in self.ignore\_dirs:

ignore\_dir\_normalized = os.path.normpath(ignore\_dir).lower()

# Проверяем точное совпадение имени директории

if ignore\_dir\_normalized == dir\_name.lower():

return True

# Проверяем вхождение в путь

if ignore\_dir\_normalized in normalized\_path.split(os.sep):

return True

return False