perplexity

파이썬 시니어 개발자 관점에서 아래의 작업을 수행해줘

첨부한 shellscript 에서 생성하도록 되어있는 파이썬 코드에서 단일 python 파일들에 있는 메서드 기능별로 class 나누고, 하나의 python class에 모든 클래스 로그해서 작동하도록 리팩토링해줘,
그냥 class 이름과 해당 class에 들어갈 method 들 리스트 적어주고,
통합할 class 이름과 통합 방식 코드는 전체 코드 제시해줘

class hello

- method1 [hello-1(args)]
- method2 [hello-2(args)]

class bye
-method1[bye-1(args)]

이러한 방식으로 class 별 메소드 리스트 정의 출력하고,
통합하는 class의 전체코드는 하나도 빠짐없이 제공해줘야해

리팩토링 계획
제공된 코드를 기능별로 모듈화하고 OOP 원칙에 맞게 클래스 구조를 재설계합니다.

클래스 구조 설계

1. `WebDriverManager` 클래스

- **역할**: 브라우저 드라이버의 전체 라이프사이클 관리
- **메서드**:
  - `setup_driver()`: 브라우저별 드라이버 초기화
  - `_cleanup_chrome_processes()`: Chrome 프로세스 정리
  - `teardown()`: 드라이버 종료 및 리소스 정리

2. `LoggerManager` 클래스

- **역할**: 로깅 시스템 초기화 및 관리
- **메서드**:
  - `setup_logging()`: 로깅 설정
  - `get_logger()`: 로거 인스턴스 반환

3. `ConfigManager` 클래스

- **역할**: 설정 파일 관리
- **메서드**:
  - `load_config()`: 설정 파일 로드
  - `validate_config()`: 설정 유효성 검증
  - `save_config()`: 설정 파일 저장

4. `AutomationExecutor` 클래스

- **역할**: 실제 자동화 작업 수행
- **메서드**:
  - `process_target()`: 대상 처리
  - `perform_action()`: 개별 액션 실행
  - `take_screenshot()`: 스크린샷 캡처

5. `MainController` 클래스

- **역할**: 전체 작업 흐름 제어
- **메서드**:
  - `run()`: 메인 실행 메서드
  - `_parse_args()`: CLI 인자 처리
  - `_setup_environment()`: 환경 설정

6. `GeminiIntegration` 클래스 (옵션)

- **역할**: Gemini API 연동
- **메서드**:
  - `generate_config()`: AI 기반 설정 생성
  - `validate_with_ai()`: AI 기반 검증

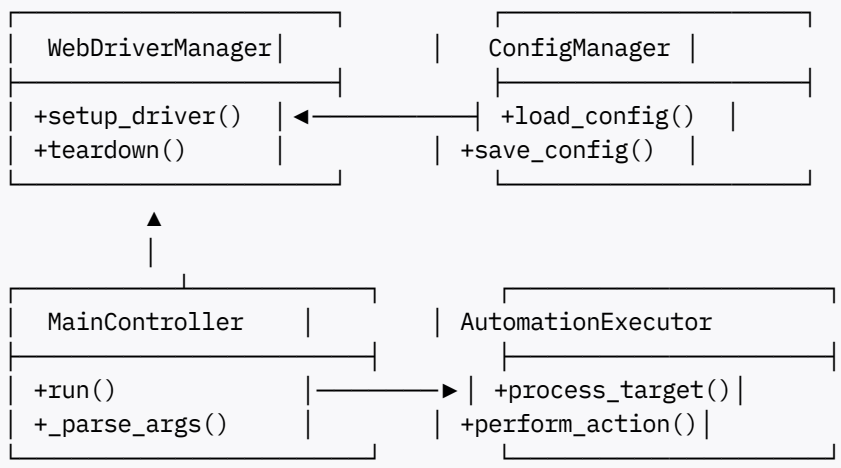
통합 클래스 코드

```
# web_automation.py
import os
import sys
import json
import logging
import argparse
from selenium import webdriver
from selenium.webdriver.chrome.options import Options as ChromeOptions

class WebDriverManager:
```

```
    def __init__(self, config):
        self.config = config
        self.driver = None
        self.user_data_dir = None

    def setup_driver(self):
        browser_type = self.config["browser"].get("type", "chrome").lower()

        if browser_type == "chrome":
            options = ChromeOptions()
            # ... [기존 setup_driver 로직 구현]

            self.driver = webdriver.Chrome(options=options)
            return self.driver

        # 다른 브라우저 지원 확장

    def teardown(self):
        if self.driver:
            self.driver.quit()
        # 리소스 정리 로직

class LoggerManager:
    @staticmethod
    def setup_logging(config):
        logger = logging.getLogger("WebAutomation")
        # ... [로깅 설정 구현]
        return logger

class ConfigManager:
    def __init__(self, config_path):
        self.config_path = config_path

    def load_config(self):
        try:
            with open(self.config_path, 'r') as f:
                return json.load(f)
        except FileNotFoundError:
            return self._create_default_config()

class AutomationExecutor:
    def __init__(self, driver, config):
        self.driver = driver
        self.config = config

    def process_target(self, target):
        # ... [대상 처리 로직 구현]

class MainController:
    def __init__(self):
        self.args = self._parse_args()
        self.config = None
        self.driver = None

    def run(self):
        self.config = ConfigManager(self.args.config).load_config()
```

```
        logger = LoggerManager.setup_logging(self.config)

        try:
            driver_manager = WebDriverManager(self.config)
            self.driver = driver_manager.setup_driver()

            AutomationExecutor(self.driver, self.config).run_workflow()

        finally:
            driver_manager.teardown()

if __name__ == "__main__":
    MainController().run()
```

클래스 다이어그램



리팩토링 이점

1. **단일 책임 원칙 준수**: 각 클래스가 명확한 역할 수행
2. **확장성 향상**: 새 브라우저/기능 추가 용이
3. **테스트 용이성**: 모듈별 독립 테스트 가능
4. **유지보수성 향상**: 코드 변경 영향 범위 최소화

이 구조는 대규모 웹 자동화 시스템의 기본 골격으로 사용 가능하며, 필요에 따라 추가 모듈을 확장하여 사용할 수 있습니다.

⁂